Rush Hour ist PSPACE-vollständig

Florian Thiemer

19. Juni 2007

Rush Hour ist ein Spiel was man auf einem quadratischem Feld spielt. Ziel des Spiels ist es ein Zielauto durch einen Ausgang am Rande des Spielfeldes zu bewegen. Autos haben die Größe von 2 oder 3 Feldern und können horizontal oder vertikal bewegt werden. Die erlaubte Bewegungsrichtung hängt von der Orientierung des Autos auf dem Spielfeld ab. Die Orientierung darf sich nicht verändern. In der Startkonfiguration blockieren sich die Autos so, dass das Zielauto nicht den Ausgang erreichen kann.
Im folgenden werden wir sehen, dass eine verallgemeinerte Version von *Rush Hour* (GRH) NP-schwer und PSPACE-vollständig ist. Dabei werden wir das Erfüllbarkeitsproblem (SAT) auf GRH polynomiell reduzieren. Anschließend zeigen wir, wie man eine beliebige Turingmaschine die zur Berechnung polynomiell viel Platz benötigt mithilfe von rückgekoppelten GRH-Schaltkreisen simmulieren kann.

1 Generalized Rush Hour (GRH)

GRH ist eine verallgemeinerte Form von *Rush Hour*. Dabei ist das Spielfeld ein Rechteck mit beliebiger Höhe und Breite. Zudem darf der Ausgang an einer beliebigen Position des Spielfeldrandes sein. Formal besteht eine Instanz von GRH aus:

- $w \in \mathbb{N}$, die Höhe des Spielfeldes
- $h \in \mathbb{N}$, die Breite des Spielfeldes
- (x_0, y_0), die Koordinaten des Ausgangs, wobei eine der beiden Variablen den Wert 0 oder w bzw. h annimmt
- $n \in \mathbb{N}$ die Anzahl der nicht-Zielautos
- $C = \{c_0, \cdots, c_n\}$ eine Menge von $n + 1$ Autos mit $c_i = (x, y, o, s)$. c_0 ist das Zielauto
  - $x \in \mathbb{N}, 1 \leq x \leq w$, die x-Koordinate des Autos
  - $y \in \mathbb{N}, 1 \leq y \leq h$, die y-Koordinate des Autos
  - $o \in \{N, S, E, W\}$, die Orientierung des Autos
  - $s \in \{2, 3\}$, die Größe des Autos

  Dabei muss C konsistent sein, in dem Sinne, dass kein Auto außerhalb des Spielfeldes stehen kann und 2 Autos sich nicht überlappen dürfen.

Eine Lösung von GRH besteht aus einer Sequenz von m Zügen. Jeder Zug besteht aus einem Autoindex i, eine Richtung und eine Distanz. Jeder Zug muss dabei Konsistent mit sich selbst und mit der Konfiguration vor dem Zug sein.
Es gibt 2 Varianten von GRH. Die Pfadversion gibt explizit einen Lösungspfad an, während das Entscheidungsproblem nur danach fragt, ob es so einen Pfad gibt.

2 Rush Hour Logik

Um Boolsche Logik darstellen zu können, werden wir eine Konfiguration für GRH so konstruieren, dass ein bestimmtes *output* Auto sich nur dann bewegen kann, wenn eine bestimmte Kombination von *input* Autos bewegt wurden.
Um Logik mittels GRH darstellen zu können, darf man nur bestimmte Bewegungen der Autos durchführen. Da in GRH keine fixen Objekte vorgesehen sind, werden wir kleine Bereiche von verankerten Autos erstellen, die notwendigerweise mit dem Spielfeldrand verbunden sind oder wieder mit einem verankertem Bereich. Um Informationen durch diese verankerten Bereiche expandieren zu lassen, bauen wir eine Triggerliniek in diesen verankerten Bereich so ein, dass sich nur die Triggerlinien um 1 oder 2 Felder bewegen können, die umschließenden verankerten Autos sich aber nicht bewegen dürfen. Damit verhindern wir, dass ein Auto sich aus einem Block entfernen kann.

3 Basic Building Blocks

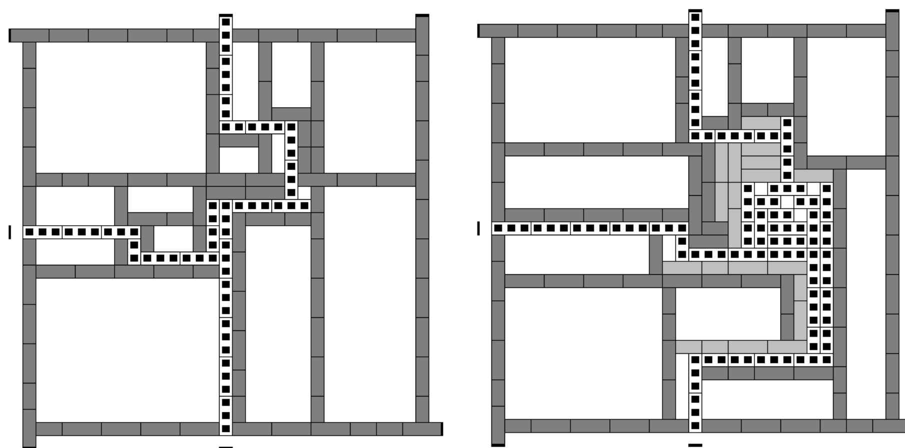


Abbildung 1: Konjunktiver und Disjunktiver Baublock

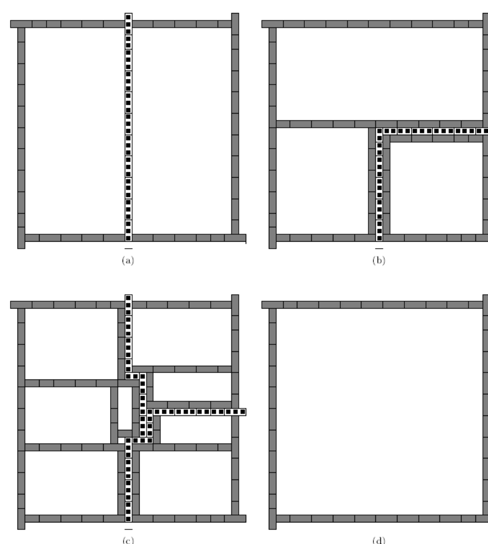


Abbildung 2: weitere Baublöcke

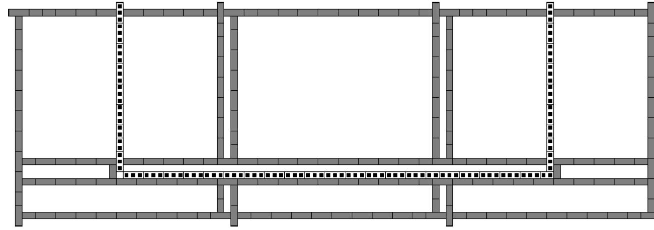


Abbildung 3: Switch

4 Logische Schaltkreise

Um nun Boolsche Logik mit GRH darstellen zu können kann man nicht einfach einen offenen und einen geschlossenen Zustand der Triggerlinien als *true* bzw. *false* festlegen. Da ein geschlossener Zustand weiterhin das Spiel blockiert. Dadurch wäre es unmöglich einen Invertierer zu bauen, der nur dann offen ist, wenn sein *input* geschlossen ist.
Um dieses Problem zu umgehen, definiert man boolsche Werte durch ein Paar von Triggerlinien. Eine Linie repräsentiert *true*, die andere *false*. In diesem Fall können beide Eingaben in anderen Blöcken Ereignisse auslösen. Nun muss man aber beachten wie man mit unsinnigen Eingaben umgeht. Es könnte sein, dass beide Linien geschlossen oder geöffnet werden. Hierzu bauen wir einen Switch der es ermöglicht, dass nur eine der beiden Triggerlinien zur selben Zeit geöffnet sein kann.
Mit A_F bezeichnen wir die Triggerlinie die *false* und mit A_T die, die *true* repräsentiert. Ein „+" und ein „-" sagt aus, ob die jeweilige Triggerline geöffnet oder geschlossen ist. Nun können wir die Zustände von A folgendermaßen aufschreiben.

$$\begin{aligned} 1 &= (A_T^+, A_F^-) \\ 0 &= (A_T^-, A_F^+) \\ - &= (A_T^-, A_F^-) \\ + &= (A_T^+, A_F^+) \end{aligned}$$

Dabei kann der Zustand „+" in unserer Konstruktion durch den Switch nie erreicht werden.
Nun können wir boolsche Logik wie folgt darstellen.

$$\begin{aligned} A \wedge B &= (A_T \barwedge B_T, A_F \veebar B_F) \\ A \vee B &= (A_T \veebar B_T, A_F \barwedge B_F) \\ \neg A &= (A_F, A_T) \end{aligned}$$

Satz 4.1. *GRH ist NP-schwer.*

Beweis. Für jede Variable eines boolschen Ausdruck γ können wir einen Switch verwenden. Wir initialisieren alle Switch mit dem „-" Zustand. γ kann nun mithilfe der GRH Schaltkreise für NOT, AND und OR dargestellt werden. Für jeden *output* setzt man den Zustand ebenfalls auf „-". Das einzige was nun übrigbleibt ist es, die einzelnen Schaltkreise mithilfe der Grundbausteine zu verbinden. Da alle Baublöcke eine konstante Größe haben und der GRH Schaltkreis linear in der Größe von γ ist, ist die Transformation von SAT nach GRH auch polynomiell im Platz- und Zeitbedarf. Zum Abschluss muss man noch das Zielauto mit dem Ausgang des konstruierten Schaltkreises versperren. Dieser letzte *output* Trigger kann nun nur dann öffnen, wenn es eine Belegung der Variablen aus γ gibt, die γ erfüllt. Lösen wir also das Entscheidungsproblem für GRH, so lösen wir auch das Entscheidungsproblem für SAT. Um eine aktuelle Belegung der Variablen zu bekommen, müssen wir uns einfach die Stellung der Switches anschauen. Dadurch kann SAT polynomiell auf GRH reduziert werden. Also ist GRH NP-schwer. □

5 GRH ist PSPACE-vollständig

5.1 GRH $\in$ PSPACE

Da PSPACE = NPSPACE ist, genügt es eine nichtdeterministische Turingmaschine M anzugeben, die zu einer gegebenen GRH-Konfiguration in $O(n^k)$ Platzbedarf, für ein $k > 0$ entscheidet, ob es eine Lösung gibt oder nicht. Mithilfe des Satzes von Savitch kann man zu M eine deterministische Turingmaschine M' konstruieren, die GRH in $O(n^{2k})$ Platzbedarf entscheidet.
Auf dem Turingband ist die Startkonfiguration einer GRH Instanz geeignet kodiert gespeichert. In jedem Zustandübergang kann sich nur ein Auto bewegen. Wenn es n Autos gibt, so gibt es höchstens $2n$ mögliche Nachfolgezustände, da sich ein Auto vorwärts, aber auch rückwärts bewegen kann. Die Nachfolgezustände probiert M einfach nichtdeterministisch aus und aktualisiert den aktuellen Zustand, indem er die Position des entsprechenden Autos einfach überschreibt. Somit löst M das Entscheidungsproblem für GRH in $O(n)$ viel Platz. Daraus folgt, dass GRH in PSPACE ist.

6 GRH ist PSPACE-schwer

Wir konstruieren dazu für jede TM $M \in$ PSPACE eine Recheneinheit aus GRH-Schaltkreisen, die für eine Eingabe x genau dann das Zielauto aus dem Spielfeld entkommen lässt, wenn x von M akzeptiert wird.
Hierzu brauchen wir eine Logik-Einheit die uns den nächsten Zustand berechnet, zwei Speicher um den Zustand abzuspeichern und einen Timer um die Speichereinheiten anzusteuern. Wir benutzen zwei Speicher nach dem „master-slave“ Prinzip. Würden wir nur einen Speicher benutzen, so könnte er beim abspeichern seinen eignen Eingang durch seinen Ausgang beeinflussen. Damit könnten wir in einen instabilen Zustand geraten.

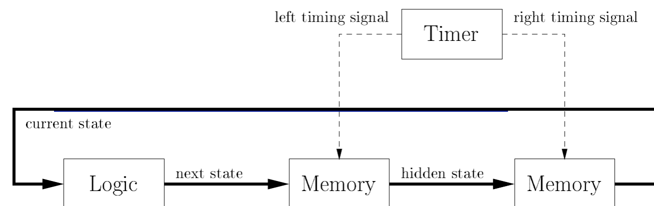


Abbildung 4: Recheneinheit mit GRH Schaltkreisen

Ein kompletter Ablauf der Recheneinheit sieht so aus:

- Die logische Einheit produziert einen Nachfolgezustand abhängig vom aktuellen Zustand
- Das linke timing-Signal ist aktiviert
- Der linke Speicher speichert seinen Input und aktualisiert den versteckten Zustand
- Das linke timing-Signal ist deaktiviert
- Das rechte timing-Signal ist aktiviert
- Der zweite Speicher speichert den Input und aktualisiert den aktuellen Zustand
- Das rechte timing-Signal ist deaktiviert.

6.1 Rückkehrbare Logik

In GRH kann man jede Sequenz von Autobewegungen auch wieder rückwärts ausführen. Dadurch entstehen aber Probleme wenn man einen Speicher realisieren will. Eine Speichereinheit kann 2 verschiedene Werte annehmen. Das bedeutet, wenn wir einen Wert im Speicher abgelegt haben, so kann er zwei mögliche Vorgängerwerte haben. Entweder der vorherige Wert ist gleich dem aktuallen oder verschieden. Das bedeutet aber auch, dass es 2 verschiedene Autobewegungen geben muss, um in den aktuellen Zustand zu gelangen. Wenn man nun sich entscheidet die Autobewegungen rückwärts auszuführen, so hat man also 2 verschiedene Möglichkeiten und kann somit in einen Zustand gelangen, der nicht der aktuelle vorherige Zustand war. Man kann also im gesamten Speicher in jeden beliebigen Zustand wechseln. Wenn wir aber ein Recheneinheit simmulieren wollen, so müssen wir verhindern, dass wir falsche Werte produzieren können. Also müssen wir einen Speicher konstruieren, der nur einen legalen vorherigen Zustand hat.

Definition 6.1. Ein logischer Ausdruck ist rückkehrbar, wenn seine Eingabe eindeutig aus der Ausgabe bestimmt werden kann. Zusätzlich kann ein beliebiger nichtrückkehrbarer Ausdruck, durch hinzufügen von weiteren In und Outputs, umgeschrieben werden, so dass er rückkehrbar ist.

Natürlich kann dann auch jeder Schaltkreis in einen äquivalenten Ausdruck überführt werden, der einen eindeutigen Inversen besitzt.
Damit konstruieren wir eine modifizierte Recheneinheit. Sie wird um rückkehrbare Logik erweitert. Es gibt nun 2 neue Einheiten, die das Ergebnis der normalen Schaltlogik einfach invertieren. Dies hat zur Folge, dass man den aktuellen Inhalt des Speichers kennt, ohne aihn auszulesen. Dadurch können wir eine Speichereinheit konstruieren die einen einheitlichen Vorgängerstatus hat.

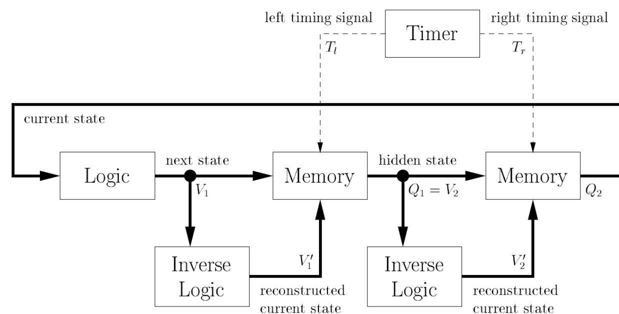


Abbildung 5: Recheneinheit mit GRH und rückkehrbarer Logik

- Das Timing Signal alterniert zwischen der linken und rechten Einheit
- Es gibt vorwärts und rückwärts - Versionen des Timing Signals
- Eine Speichereinheit nimmt ihren vorherigen Zustand an, wenn es ein rückwärts Signal des Timers erhält
- Jeder Speicherinput hängt vom anderen Speicheroutput ab
- Kein Speicher kann seinen eigenen Input verändern.

Zusammengefasst bedeutet dass, solange die Speicher und der Timer rückkehrbar und invertierbar sind, hat die Recheneinheit einen einzigen eindeutigen Vor- und Nachfolgezustand.

6.2 GRH Zustandsgraphen

Um die folgenden Schaltkreise leichter analysieren zu können, konstruieren wir zu jedem Schaltkreis einen Graphen, so dass jeder Knoten einen zulässigen Zustand und die Kanten zwischen den Knoten zulässige Übergänge repräsentieren. Wenn wir die Eigenschaften des Graphen untersuchen, können wir gleichzeitig die Eigenschaften des GRH Schaltkreises verifizieren.
Der aktuelle Zustand einer GRH Konfiguration besteht aus den Position der Triggerlinien in den Grundbausteinen aus denen sich ein GRH Schaltkreis zusammensetzt. Die Zustandsbeschreibung wird als binärer Vektor beschrieben. „1“ bedeutet dass die Triggerlinie geöffnet ist und „0“ dass sie geschlossen ist. Man muss alle Zustände aus den Graphen entfernen, die aufgrund der Konstruktion der Grundbausteine unzulässig sind. Anschließend verbindet man die Zustandsknoten, die sich nur um ein Bit unterscheiden.

6.3 Simple Latch

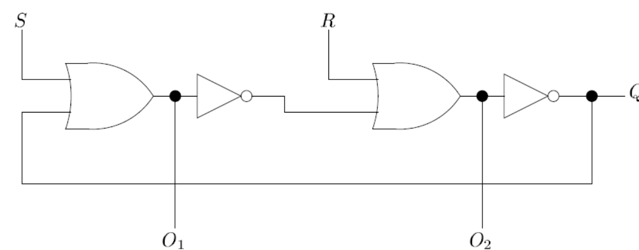


Abbildung 6: Simple Latch

Mit dem in 6 gezeigten Schaltkreis kann man 1 Bit abspeichern. Der GRH Schaltkreis ist strukturell identisch zum klassischen Latch, aber seine Funktionsweise ist komplett anders. Um die Funktionsweise zu verstehen, betrachten wir den Zustandsgraphen des Simple Latch.

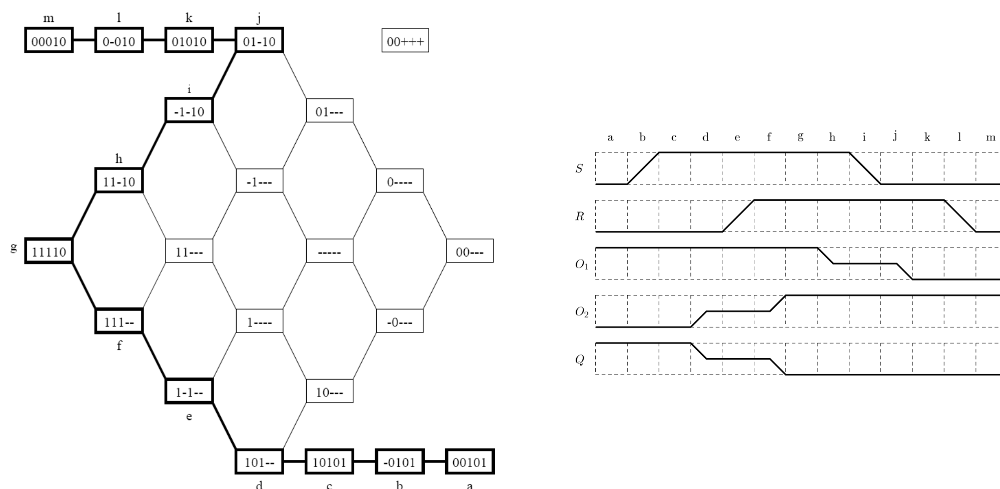


Abbildung 7: Zustandübergangsgraph des Simple Latch

Betrachten wir den Graphen, können wir folgendes feststellen:

- Der Output kann sich nicht ändern, wenn die Eingänge bei 0 festgehalten sind.
- Einen Eingang zu belegen, kann niemals in einer Änderung des Ausganges enden

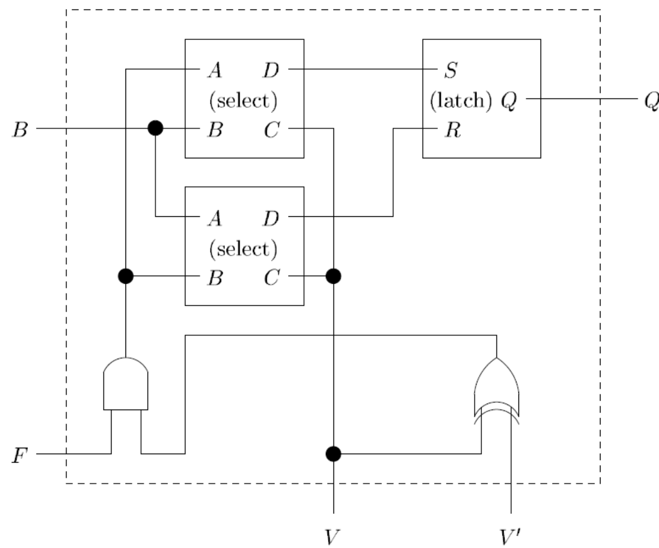


Abbildung 8: rückkehrbarer Latch

Damit besteht die einfachste Sequenz um den Output zu ändern aus einem „front“ und einem „back“ Puls wie er in der Tabelle eingezeichnet ist.

6.4 rückkehrbarer Latch

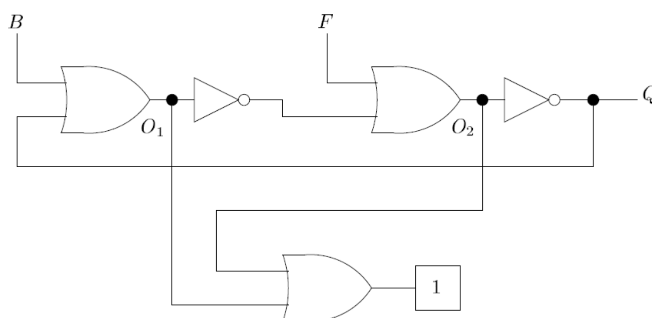
Der Simple Latch alleine reciht noch nicht aus um unsere Recheneinheit in dem Sinne zu erstellen, dass jeder Speicher einen eindeutigen vorherigen Zustand hat. Hierzu konstruieren wir uns einen rückkehrbaren Latch wie folgt.

Ein rückkehrbarer Latch ist in Beispiel 8 gezeichnet. Er nutzt zwei selection Schaltkreise, ein extra AND und ein Exklusiv-Oder. Der Selectionschaltkreis funktioniert wie folgt: Wenn C 0 ist, dann nimmt D den Wert von A an, sonst nimmt D den Wert von B an.
Für die Eingänge nehmen wir an, dass F ein front Puls und B einen back Puls durchläuft. V ist der nächste Zustand des Latch und $V^{'}$ der rekonstruierte aktuelle Zustand des Latch. Wenn V gleich 1 ist, so wird der back Puls unverändert zum S Eingang des einfachen Latch weitergeleitet. Aber wenn V gleich 0 ist, so wird der back Puls unverändert zum R Input weitergeleitet. Ob wir nun den Inhalt des Latches verändern oder nicht, können wir somit über den front Puls steuern. Dazu unterdrücken wir ihn gegebenenfalls, so dass nur der back Puls an den Simple Latch geleitet wird und den aktuellen Wert nicht verändern kann (siehe Abbildung 7).
Wir benutzen den wiederhergestellten aktuellen Zustand des Latch V' zusammen mit V um zu bestimmen, ob ein front Puls durchgelassen oder unterdrückt werden soll. Wenn V und V' verschieden sind, dann wird der front Puls durchgelassen und garantiert für den Simple Latch eine Abfolge wie in 7. In diesem Fall wird der Output entweder von 0 auf 1 oder von 1 auf 0 geändert werden. Sind allerdings V und V' identisch, dann wollen wir das sich der Output nicht ändert. Dazu brauchen wir das AND und XOR Gate. Mit dem XOR Gate überprüfen wir, ob der aktuelle und nächste Zustand gleich sind. Wenn es so ist, wird der front Puls mit dem AND Gate unterdrückt, und nach dem Zustandsgraphen des Simple Latch würden wir den Ausgang nicht verändern.
Um zu sehen, dass man diesen Latch invertieren kann, betrachte wir den Fall wenn die Timing Signale verkehrt herum gesendet werden kurz nachdem der Latch beschrieben wurde. Die werte von V und V' sind also noch nicht verändert. In dem umgekehrten Fall ist also der initialisierende

Puls auf B und der Nachfolgende auf F. Solange V und Q gleich sind, tut der initialisierende Puls nichts bedeutendes, da das XOR und AND Gate den nachfolgenden Puls unterdrücken würden, was bedeutet, dass der vorherige Zustand rekonstruiert wird.

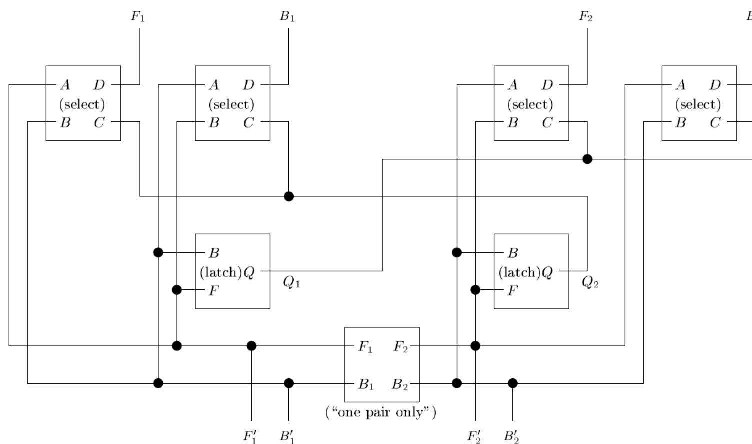
6.5 Einfacher Oszillator

Wir haben nun einen Speicher konstruiert, der Werte abspeichern kann und auch einen eindeutigen Vorgängerzustand hat. Wir müssen jetzt nur noch ein Gerät konstruieren, dass uns ein Pulspaar generiert um die Speichereinheiten anzuseteuern. Hierzu benutzen wir wieder den Simple Latch und erweitern ihn durch ein Oder-Gatter, dessen Ausgang fest auf 1 gelegt ist. Somit können wir sicherstellen, dass die beiden Eingänge nicht gleichzeitig den Wert „0“ oder „-“ haben. Um



die Funktionsweise zu verstehen betrachten wir einfach wieder den Zustandsgraphen des Simple Latch (6). Da einer der beiden Variablen O_1 oder O_2 den Wert 1 haben muss, so kann der Einfache Oszillator im Zustandsgraphen nur die fett-markierte Linie ablaufen. Ändern wir also den Wert des Ausgangs, so erzeugen wir bei F und B einen front bzw. back Puls, die wir dazu benutzen können, um einen Speicher anzusteuern.
Der einfache Oszillaotr ist alleine aber ungenügend um die Speicher anzusteuern. Haben wir einmal einen front und back Puls erzeugt, so müssen wir sie anschließend wieder in umgekehrter Reihenfolge erzeugen, da wir im Zustandsgraphen wieder zurückgehen müssen. Das würde die Speichereinheiten aber wieder zurücksetzen und man kommt nicht voran. Desweiteren haben wir zwei Speicher die seriell geschaltet sind. Wir brauchen also zwei Pulspaare die hintereinander kommen um beide Speicher anzusetuern.

6.6 Dual Timer



Hierzu benötigen wir einen weiteren Hilfsschaltkreis, den wir als *one-pair* Schaltkreis bezeichnen. Er funktioniert so, dass entweder F_1 und B_1 den Wert 0 haben, oder F_2 und B_2. Es kann also immer nur einer der beiden Paare zu einem Zeitpunkt aktiv sein.
Der Dual Timer funktioniert so, dass er den Ausgang eines Oszillators nimmt, um eventuell die beiden erzeugten Pulspaare des anderen Oszillators zu vertauschen. Damit können wir dann eine endlose Folge von Pulspaaren erzeugen.
Da nur eines der beiden Pulspaare aktiv zu einem Zeitpunkt sein können, so kann man nicht gleichzeitig Q_1 und Q_2 ändern. Es gibt also insgesamt 8 Fälle. Wobei 4 davon symmetrisch zu den anderen sind. Dabei werden die Pulspaare nur in umgekehrter Reihenfolge erzeugt.

$Q_1 : 0 \to 1, Q_2 : 0 \to 0$

Hier dürfen nur $F_1^{'}$ und $B_1^{'}$ ihre Werte ändern. Da Q_2 die ganze Zeit 0 ist, so korrespondieren die Ausgänge F_1 und B_1 zu $F_1^{'}$ und $B_1^{'}$.

$Q_1 : 1 \to 1, Q_2 : 0 \to 1$

Hier können sich nur $F_2^{'}$ und $B_2^{'}$ verändern. Da Q_1 auf 1 gesetzt ist, so korrespondieren F_2 und B_2 zu $F_2^{'}$ und $B_2^{'}$. Bis hierhin haben wir also 2 Pulspaare erzeugt, die einen Rechenschritt der Recheneinheit darstellen soll um beide Speicher anzuseteuern.

$Q_1 : 1 \to 0, Q_2 : 1 \to 1$

Dies erzeugt einen gleichen Output wie im ersten Fall, aber auf unterschiedlicher Funktionsweise. Da Q_1 auf 1 gesetzt ist, sind die Signale von $F_1^{'}$ und $B_1^{'}$ vertauscht. Da aber Q_2 auf 1 gesetzt sind, wird das Pulspaar wieder umgedreht und somit haben wir die gewünschte Ausgabeform.

$Q_1 : 0 \to 0, Q_2 : 1 \to 0$

Ähnlich wie im vorherigen Fall werden die Eingänge bei Q_2 in umgekehrter Reihenfolge ausgegeben. Da Q_1 aber den Wert 0 hat, wird das Pulspaar wieder richtig herum gedreht.
Nun sind wir wieder in der Ausgangssituation und können weitere Pulspaare erzeugen.

Satz 6.2. *GRH ist PSPACE-schwer*

Beweis. Wir haben gezeigt, dass wir mit GRH beliebige rückgekoppelte Schaltkreise erzeugen können. Es ist aber noch nicht klar, wieviel Aufwand man investieren muss, damit alle Speichereinheiten invertierbar sind. Wenn wir also eine normale Turingmaschine M auf einer Turingmaschine M' simmulieren, die mit rückkehrbarer Logik funktioniert, wieviel zusätzlichen Speicher benötigt dann M' um jeden Schritt rückkehrbar zu machen? C. H. Bennet hat gezeigt [2], dass man für jedes $\epsilon > 0$ M mit Zeit- und Platzbedarf T und S auf M' mit Zeit $O(T^{1+\epsilon})$ und Platzbedarf $O(SlogT)$ simmulieren kann. Also können wir auch eine GRH Konfiguration angeben, die M simmuliert und dass Zeit- und Platzbedarf polynomiell in der Größe von S und T liegt. □

Literatur

[1] Gary W. Flake, Eric B. Baum: Rush Hour is PSPACE-complete, or „Why you should generously tip parking lot attendants“

[2] C. H. Bennet: Time/Space trade-offs for reversible computation. *SIAM J. Computing*, 18(4):766-776, 1989.