VU Advanced Digital Design

Homework 2

Bernhard Urban

Matr.Nr.: 0725771

lewurm@gmail.com

21. Mai 2011

Inhaltsverzeichnis

1 Micropipeline **2**
1.1 Vorteile der Micropipeline . . . 2
1.2 Simulation . . . 2
1.3 Volle und leere Pipeline . . . 2

2 Muller C-Element **5**
2.1 C-Element Implementierungen . . . 5
2.1.1 Martin . . . 5
2.1.2 Sutherland . . . 6
2.1.3 van Berkel . . . 7
2.2 Delay Insensitivity . . . 7
2.3 LUT Implementierung . . . 8

3 XNOR-Gate for QDI **9**
3.1 Implementierung mit SR-Latch . . . 9
3.2 Implementierung mit D-Latch . . . 11

4 Protocols **12**

Literatur **13**

1 Micropipeline

Unter einer Micropipeline versteht man einen elastischen und eventgesteuerten FIFO.

elastisch: Es sind nur immer so viele Stages aktiv wie nötig. Zum Beispiel, können Daten die an eine leere Pipeline angelegt werden, "sofort" durchpropagiert werden. Ist die Pipeline bereits befüllt, so können neue Daten bis zur letzten leeren Stage vor der ersten belegten Stage propagiert werden.

eventgesteuert: Im Gegensatz zum synchronen Design, werden hier die Stages mit Events, d.h. Request und Acknowledge, gesteuert.

FIFO: **F**irst **I**n **F**irst **O**ut ⇒ die Eingabereihenfolge entspricht der Ausgabereihenfolge.

1.1 Vorteile der Micropipeline

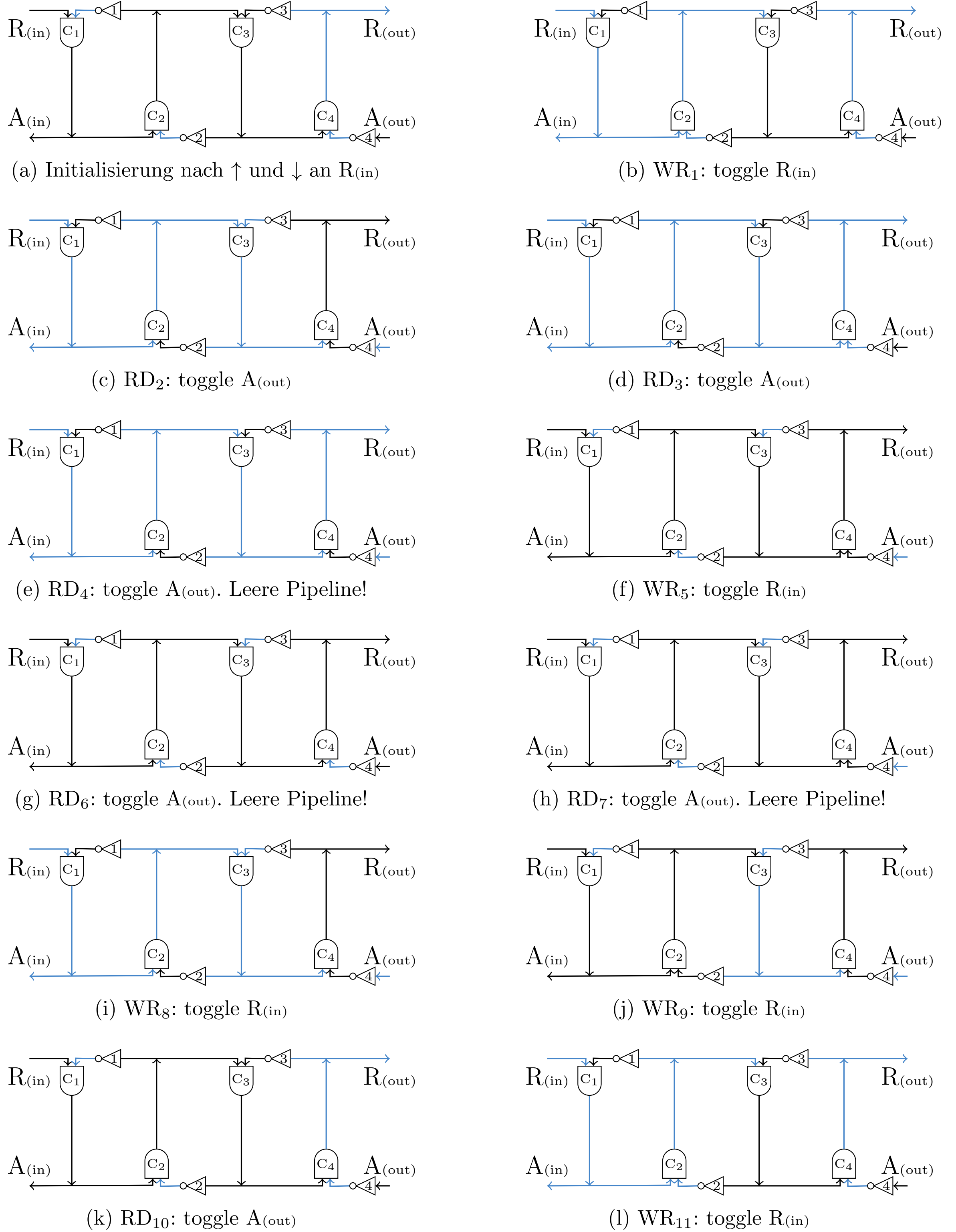
- Das Konzept von Transition Signalling sei einfacher zu designen und zu verstehen. Außerdem ist es ähnlich zum Softwaredesign, und daher "bereits bekannt".
- Transition Signalling doppelt so schnell, da nur noch auf Flanken (steigend sowohl als fallend) getriggert wird.
- Man kann leicht mehrere kleinere Strukturen zu einem grossen Block verbinden, da die Schnittstelle sehr einfach gestaltet wurde ⇒ Composability.
- Durch Composability ist es weiters möglich spezifische Teile eines Systems *einfach* zu ersetzen, ohne auf Seiteneffekte achten zu müssen.
- Kontrollsignale sind (abgesehen vom Inverter und dem Delay) in beide Richtungen der Pipeline gleich (simples Design!).
- Clockrouting entfällt. Gewissermassen ist das auch eine Energieeinsparung, da die Clock nicht in der ganzen Schaltung (global) verteilt werden muss.

1.2 Simulation

Ich habe die Variante "Simulation per Hand" gewählt. Das Ergebnis ist in Abbildung 1 ersichtlich. Dabei entspricht schwarz logisch 0 und blau logisch 1. Bei der Initialisierung wird von einer Pipeline ausgegangen, in der initial die Ausgänge der C-Elemente logisch 0 (d.h. schwarz) sind.

1.3 Volle und leere Pipeline

Ausgehend von der Pipeline in Abbildung 1b, legen wir noch ein weitere Transition an. Das Ergebnis ist in Abbildung 2a ersichtlich. Wie man erkennen kann, hat nun jede weitere Transistion auf $R_{(in)}$ keine Auswirkung, solange bis $A_{(out)}$ getoggelt wird. Unter anderem hat eine volle Pipeline sozusagen auch ein Muster: Die Ausgänge der



(a) Initialisierung nach ↑ und ↓ an R(in)

(b) WR_1: toggle R(in)

(c) RD_2: toggle A(out)

(d) RD_3: toggle A(out)

(e) RD_4: toggle A(out). Leere Pipeline!

(f) WR_5: toggle R(in)

(g) RD_6: toggle A(out). Leere Pipeline!

(h) RD_7: toggle A(out). Leere Pipeline!

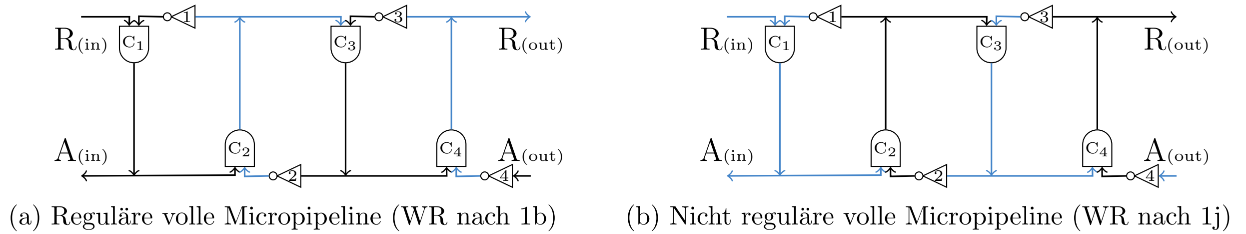
(i) WR_8: toggle R(in)

(j) WR_9: toggle R(in)

(k) RD_{10}: toggle A(out)

(l) WR_{11}: toggle R(in)

Abbildung 1: Simulation einer 4-Stage Micropipeline



(a) Reguläre volle Micropipeline (WR nach 1b) (b) Nicht reguläre volle Micropipeline (WR nach 1j)

Abbildung 2: Volle Pipelines

auf einander folgenden C-Elemente haben alternierende Werte (hier: 0101) und $R_{(out)} \neq A_{(out)}$.

Leere Pipelines sind leider nicht mehr so einfach im statischen Sinne (d.h. "durch Hinschauen") zu bestimmen. Wie ich in den Abbildungen beschriftet habe, sind 1e, 1g und 1h leere Pipelines. Man könnte nun behaupten, eine Pipeline sei leer, wenn alle Ausgänge der C-Elemente den selben Wert annehmen (hier 1111 bzw. 0000). Ein Gegenbeispiel zur dieser Behauptung findet man in Abbildung 1f; in dieser Pipeline befindet sich genau ein Element. Als weitere Restriktion könnte man die Bedingung $R_{(out)} = A_{(out)}$ einführen. Somit wären unsere leeren Pipelines 1e und 1g eindeutig bestimmt. Nun gibt es noch 1h. *Eigentlich* ist diese Pipeline auch leer, erfüllt aber nicht unsere Bedingungen, d.h. mit diesen Bedingungen können wir nicht *alle* leere Pipelines bestimmen. "Lustiger" wird das ganze noch, wenn man 1f mit 1h vergleicht, denn hier stellen wir fest, dass eine leere Pipeline *genauso* wie eine Pipeline mit einem Element ausschaut $\Rightarrow$ wir können (statisch) keine Pipeline mit genau einem Element bestimmen.

Auf den Zustand könnte man im dynamischen Sinne durchaus schliessen, indem man die Historie der Pipeline betrachtet. Es stellt sich allerdings die Frage ob man das will. Weiters bleibt offen, ob solch eine Situation wie in 1h (d.h. RD auf eine leere Pipeline) in der Praxis überhaupt vorkommt. Dieser Schritt hat nicht nur den Nachteil, dass man sich die Bestimmung des Zustandes der Pipeline erschwert, sondern auch eine Stage blockiert: Macht man, ausgehend von 1j, nochmal ein WR, so ergibt sich eine "volle" Pipeline, wie in 2b abgebildet, in der sich aber nur drei Elemente befinden...

D.h. würde man 1h (per Def.) verbieten, dann wären leere Pipelines und auch einelemente Pipelines (alle Ausgänge der C-Elemente sind gleich und $R_{(out)} \neq A_{(out)}$) *eindeutig* und *statisch* bestimmbar.

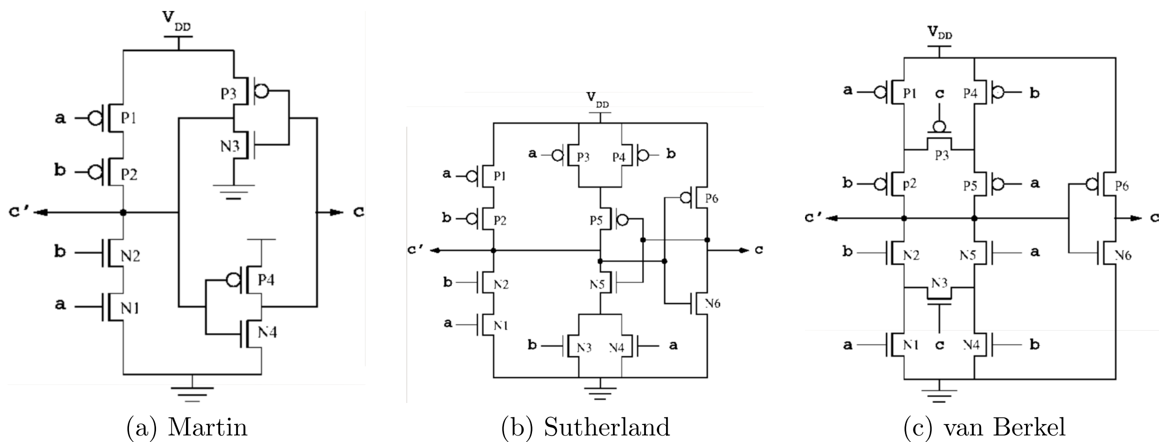
Nachtrag: Ein Kollege wies mich darauf hin, dass in [Sut89][Figure 4] festgelegt wird, dass ausschliesslich ein Acknowledge auf ein Request folgen darf. Somit ist der Schritt in Abbildung 1h tatsächlich ungültig!

2 Muller C-Element

Das Muller C-Element bietet prinzipell diese Funktionalität:

A	B	Z
1	1	1
0	0	0
1	0	gespeicherter Wert
0	1	gespeicherter Wert

Wie man schnell sieht, kann man das Muller C-Element nicht rein kombinatorisch realisieren, sondern benötigt einen internen Speicher. Ein solches C-Element kann nun verschiedenst implementiert werden, drei Möglichkeiten aus der Vorlesung werden im Folgenden disktutiert.



(a) Martin (b) Sutherland (c) van Berkel

Abbildung 3: Implementierungen für das Muller C-Element

2.1 C-Element Implementierungen

2.1.1 Martin

Siehe Abbildung 3a. Funktional betrachtet, kann man diese Schaltung in zwei Abschnitte teilen: P_1 und P_2 bzw. N_1 und N_2 sind dafür zuständig um neue Werte in die Storage Loop einzubringen. P_3, N_3, P_4 und N_4, zwei entgegengerichtete Inverter, bilden diese Storage Loop. Nun können wir verschiedene Fälle betrachten:

$a = b = 0$: P_1 und P_2 schalten durch, dementsprechend übernimmt c' den Wert logisch 1 (verbunden mit V_{DD}). Dadurch wird N_4 geschalten und schlussendlich wird c auf log. 0 gesetzt (da N_4 mit GND verbunden ist). Entsprechend stellt sich die Storage Loop ein.

Man beachte jedoch folgende Ausgangssituation: $c = 1$ ($\Rightarrow c' = 0$). N_3 wird hier von c geschalten, deshalb wird c' von GND getrieben. Da aber von P_1 und P_2 nun

V_{DD} auf die selbe Leitung geschaltet wird, entsteht (kurzfristig) ein Kurzschluss. Dass die Schaltung auch tatsächlich funktioniert, muss P_1 und P_2 stärker treiben als N_3.

$a = b = 1$: Analog zu $a = b = 0$.

$a = 1 \wedge b = 0 \wedge c = 0$: Da $a \neq b$, soll c den alten Wert, in diesem Fall 0, am Ausgang halten. Betrachtet man die Storage Loop, aktiviert $c = 0$ P_3 und schaltet deshalb V_{DD} auf c'. Dieses wiederrum schaltet N_4, welcher mit GND verbunden ist, und demzufolge bleibt c weiterhin log. 0.

$a = 1 \wedge b = 0 \wedge c = 1$: Analog zu $c = 0$.

$a = 0 \wedge b = 1$: Analog zu $a = 1 \wedge b = 0$.

Vorteile: Braucht wenige Transistoren und ist einfach zu verstehen.

2.1.2 Sutherland

Siehe Abbildung 3b. Diese Schaltung ist recht ähnlich zur Implementierung von Martin, nur dass er “linke” Teil (d.h. die Logik zum Ansteuern der Storage Loop) erweitert wurde (vgl. P_3 und P_4 bzw. N_3 und N_4), um den angesprochenen Kurzschluss von Martin zu verhindern. Außerdem wurde die Storage Loop (grafisch) anders angeordnet. Wir wollen wieder ein paar Fälle betrachten:

$a = b = 1 \wedge c = 0$: N_1, N_2, N_3 und N_4 werden durchgeschalten. Dies verursacht, dass $c' = 0$ wird[1], welches dann P_6 schaltet, der wiederrum mit V_{DD} verbunden ist und $c = 1$ bewirkt.

$c = 1$ selbst verursacht dann, dass N_5 aktiviert wird, welches (durch die geschaltenen N_3 und N_4) GND schaltet. Dies schaltet weiters P_6 $\Rightarrow$ die Storage Loop hat den Wert übernommen.

$a = 1 \wedge b = 0 \wedge c = 0$: In diesem Fall soll die Storage Loop den Wert liefern. Prinzipell werden durch a und b, N_4 und P_4 geschalten. Da $c = 0$ ist, wird P_5 aktiviert $\Rightarrow$ $c' = 1 \Rightarrow$ schaltet N_6, welches mit GND verbunden ist $\Rightarrow c$ bleibt log. 0.

Andere Fälle kann man sich analog zu den gezeigten Fällen bzw. ähnlich zur Schaltung von Martin durchdenken. Der Vorteil dieser Schaltung besteht darin, dass sie die Schwächen in der Realisierung von Martin, nämlich den Kurzschluss, beseitigt. Dadurch sind auch keine Transistoren mit unterschiedlichen Treiberstärken nötig.

[1] Zu diesem Zeitpunkt ist V_{DD} von P_5 durch P_3 und P_4 schon “abgekoppelt” und es kann deshalb nicht, wie bei Martin, zu einem Kurzschluss kommen (angenommen die Delays passen...)

2.1.3 van Berkel

Siehe Abbildung 3c. Diese Schaltung unterscheidet sich deutlich von den anderen, so sind die zwei vorher angesprochenen "Teile" nun ineinander verflossen, d.h. die Storage Loop ist nun nicht mehr explizit zu erkennen. Anhand von zwei Beispielen wollen wir hier die Funktionsweise der Schaltung zeigen:

$a = b = 0 \wedge c' = 0 \wedge c = 1$: P_1 und P_2 werden geschalten, aber auch P_4 und P_5. Da $c = 1$ ist, wird N_3 aktiviert, ist aber in diesem Fall nicht von Bedeutung. Da nun c' mit V_{DD} verbunden ist, wird N_6 geschalten und ist somit mit GND verbunden $\Rightarrow c = 0$.

$a = 1 \wedge b = 0 \wedge c' = 0 \wedge c = 1$: In diesem Fall soll der gespeicherte Wert ausgegeben werden. Durch $c = 1$ wird N_3 geschalten, gleichzeitig wird wegen $a = 1$ N_1 und N_5 aktiviert, die über N_3, GND mit c' verbinden. $c' = 0$ aktiviert daher P_6 und schaltet daher $c = 1$.

Ein Vorteil dieser Schaltung ist, dass nur noch ein Inverter sozusagen explizit enthalten ist. Dadurch kann man ein besseres Propagation Delay (bei gleichen Energieverbrauch) als bei den anderen Implementierungen erreichen (laut [SEE96]). Außerdem hat man auch, wie schon bei Sutherland, nicht das Problem mit den unterschiedlichen Treiberstärken.

2.2 Delay Insensitivity

Schon alleine wegen den vier Eingängen an jeder Schaltung (Stichwort PVT Variations) und den Gabelungen in den Schaltungen, kann man nicht auf Delay Insensitivity schließen. Trotzdem wollen wir es im Folgenden diese Einschränkungen als gelöst betrachten, um andere Probleme die Delay Insensitivity verletzen, aufzuzeigen.

Martin

Man betrachte folgende Situation, $c = 0 \Rightarrow c' = 1$: N_4 ist daher geschalten und stellt GND zu c durch. Nun wird $a = b = 1$ gesetzt und c' wird deshalb mit GND verbunden. Dieses Signal aktiviert P_4 und schaltet N_4 aus. Um nun einen Kurzschluss und, viel wichtiger, einen nicht definierten Zustand am Ausgang zu vermeiden, muss folgendes gelten:

$$T_{P_4.\text{einschalten}} > T_{N_4.\text{auschalten}}$$

Wenn wir uns ähnliche Beispiele durchdenken, kommen wir auf die allgemeine Eigenschaft, dass die Einschaltzeit der Transistoren länger brauchen muss als die Zeit zum Ausschalten. Diese Bedingung verletzt daher Delay Insensitivity.

Sutherland

Hier gilt eine ähnliche Bedingung wie bei Martin, nur ist die Einschränkung hier durch die zusätzlichen Transistoren (P_3 und P_4 bzw. N_3 und N_4) ein wenig entschärft wird,

A	B	Z'	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabelle 1: Tabellarische Darstellung zur Implementierung mittels LUT

da hier die Transistoren einen zusätzlichen "Buffer" haben, bis quasi die Leitungen die Werte von P_1 und P_2 bzw. N_1 und N_2 übernommen haben. Nichtsdestotrotz wird Delay Insensitivity verletzt.

van Berkel

Der Vorteil in dieser Schaltung besteht darin, dass hier nur ein Inverter explizit enthalten ist. Nach wie vor besteht aber das Problem mit den Schaltzeiten: Für P_6 und N_6 muss wieder die oben genannte Bedingung gelten um nur gültige Ausgaben auf c zu erhalten, deshalb verletzt auch diese Schaltung letztenendes Delay Insensitivity.

2.3 LUT Implementierung

Eine Muller C-Element Implementierung mittels Lookup-Tabelle, kann man sich wie in Tabelle 1 angegeben überlegen. Z' ist dabei die Rückführung von Z und gleichzeitig der kritische Teil dieser Umsetzung bezüglich Delay Insensitivity. Wenn sich die Eingänge von A oder B schnell ändern, so kann dies zu Problemen führen wenn das Propagation Delay von Z' nicht kurz genug ist.

A	A_0	A_1	B	B_0	B_1	Z	Z_0	S_0	R_0	Z_1	S_1	R_1
L	1	0	L	1	0	H	0	0	1	1	1	0
L	1	0	H	0	1	L	1	1	0	0	0	1
L	1	0	l	0	0	⋆	–	–	–	–	–	–
L	1	0	h	1	1	⋆	–	–	–	–	–	–
H	0	1	L	1	0	L	1	1	0	0	0	1
H	0	1	H	0	1	H	0	0	1	1	1	0
H	0	1	l	0	0	⋆	–	–	–	–	–	–
H	0	1	h	1	1	⋆	–	–	–	–	–	–
l	0	0	L	1	0	⋆	–	–	–	–	–	–
l	0	0	H	0	1	⋆	–	–	–	–	–	–
l	0	0	l	0	0	h	1	1	0	1	1	0
l	0	0	h	1	1	l	0	0	1	0	0	1
h	1	1	L	1	0	⋆	–	–	–	–	–	–
h	1	1	H	0	1	⋆	–	–	–	–	–	–
h	1	1	l	0	0	l	0	0	1	0	0	1
h	1	1	h	1	1	h	1	1	0	1	1	0

Tabelle 2: XNOR Implementierung mit Hilfe von SR Latches

3 XNOR-Gate for QDI

3.1 Implementierung mit SR-Latch

In Abbildung 4 sind Wahrheitswerte für FSL im Allgemeinen (4a) und für das gesuchte XNOR-Gate (4b) abgebildet.

	$rail_0$	$rail_1$
L	1	0
H	0	1
l	0	0
h	1	1

(a) für FSL

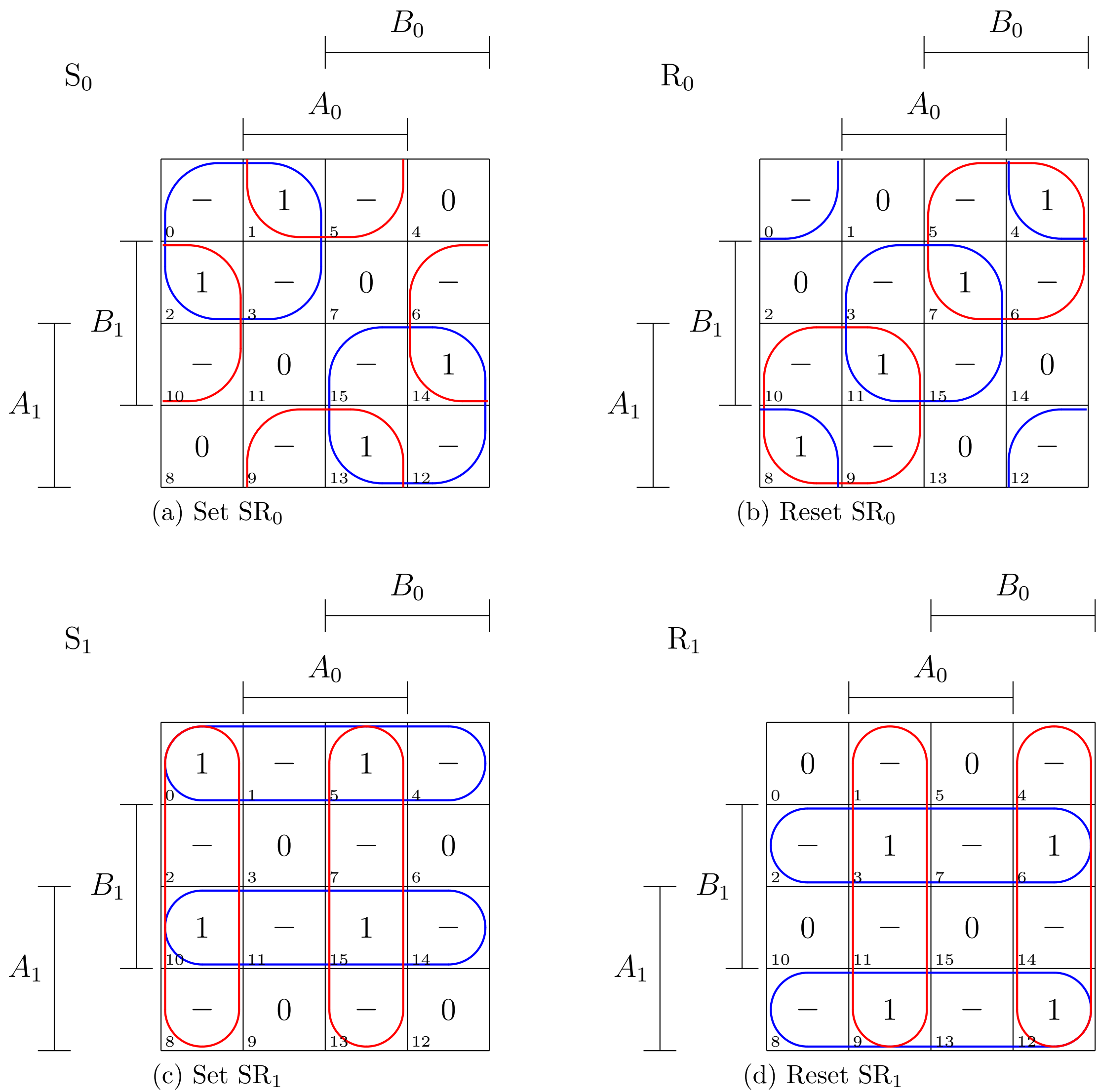
B \ A	H	L	h	l
H	H	L	⋆	⋆
L	L	H	⋆	⋆
h	⋆	⋆	h	l
l	⋆	⋆	l	h

⋆ . . . letzter Wert

(b) für das XNOR-Gate

Abbildung 4: Wahrheitstabellen

Daraus folgt die Tabelle 2. Da der Ausgang ebenfalls wieder in FSL sein muss, benötigt man zwei SR Latches, die entsprechend beschaltet werden müssen. Nun kann man die Terme für jeden Eingang der SR Latches mittels KV-Diagramme (vgl. Abbildung 5) vereinfachen, um die minimalen Terme zu ermitteln. Schließlich fassen wir die Ergebnisse



(a) Set SR_0 (b) Reset SR_0

(c) Set SR_1 (d) Reset SR_1

Abbildung 5: KV-Diagramme zur Implementierung von XNOR.
"–" ist dabei als *Don't Care* zu verstehen

aus Abbildung 5 zusammen:

$$
\begin{aligned}
S_0 &= ((A_1 \wedge B_0) && \vee(\neg A_1 \wedge \neg B_0)) && \wedge((A_0 \wedge \neg B_1) && \vee(\neg A_0 \wedge B_1)) \\
R_0 &= ((A_0 \wedge B_1) && \vee(\neg A_0 \wedge \neg B_1) && \wedge((A_1 \wedge \neg B_0) && \vee(\neg A_1 \wedge B_0)) \\
S_1 &= ((A_0 \wedge B_0) && \vee(\neg A_0 \wedge \neg B_0)) && \wedge((A_1 \wedge B_1) && \vee(\neg A_1 \wedge \neg B_1)) \\
R_1 &= ((A_0 \wedge \neg B_0) && \vee(\neg A_0 \wedge B_0)) && \wedge((A_1 \wedge \neg B_1) && \vee(\neg A_1 \wedge B_1))
\end{aligned}
$$

Aus den KV-Diagrammen ist übrigens auch schön ersichtlich, dass SIC-Hazards nicht möglich sind.

3.2 Implementierung mit D-Latch

f und g entsprechen offensichtlich S_0 und S_1 aus der vorigen Teilaufgabe. Das Enablesignal lässt sich mit einem Paritätsblock einfach bestimmen, aufgrund der "Phaseneigenschaft" von FSL. Zustände die sich in der gleichen Phase befinden (vgl. Abbildung 4a), ergeben eine gerade Anzahl von Einsern (demzufolge muss man einen "Odd-parityblock" für die Implementierung wählen).

Bezüglich des Delays ist die SR-Latch Variante relativ unempfindlich – wenn man davon ausgeht dass die Schaltung möglichst ohne Glitch gebaut wird – da sich auf jeden Pfad die gleiche Anzahl von AND- und OR-Gattern wieder findet. Man muss allerdings darauf achten, dass die Set- bzw. Resetleitung niemals gleichzeitig logisch 1 werden.

Bei der D-Latch Variante muss man darauf achten, dass der Parityblock entsprechend verzögert wird (dieser hat offensichtlich einen kürzeren Pfad als f respektive g), d.h. $T_{Parity} > \max(T_f, T_g) + T_{Setup}$ muss gelten, um keinen Glitch beim Anlegen neuer Daten zu erzeugen. Außerdem muss darauf geachtet werden, dass die Holdbedingung nicht verletzt wird $\min(T_f, T_g) > T_{Hold}$. Aufgrund dieser "nervigen" Eigenschaften würde ich die SR-Latch Variante vorziehen.

A_0	A_1	Z
0	0	NULL
0	1	TRUE
1	0	FALSE
1	1	illegal

Tabelle 3: Null Convention Logic

4 Protocols

Im folgenden wird die Datensequenz {01,00,00} mit verschiedenen Protokollen mit Hilfe eines Timingdiagrammes dargestellt und ggf. Delaybedingungen eingeführt, um ein korrektes Arbeiten des Protokolles zu garantieren.

- **4-phase bundled data 6a**: Folgende Bedingungen müssen eingehalten werden. Zunächst muss darauf geachtet werden, dass die anliegenden Daten vom Empfänger rechtzeigt "gelatcht" werden. D.h. das Signal von Acknowledge und die Zeit durch die Logikwolke müssen grösser als die Holdtime der Latches am Receiver sein:

  $$T_{\text{ACK}} + T_{\text{Data}} > T_{\text{Hold}}$$

  Diese Bedingung ist aber nicht besonders schmerzhaft, weil man das $>$ in der Regel durch $>>$ ersetzen kann. Die nächste Bedingung ist, dass der Request länger braucht als die Logikwolke[2]:

  $$T_{\text{REQ}} > T_{\text{Data}}$$

- **2-phase bundled data 6b**: Gleichen Bedingungen wie oben.

- **4-phase dual-rail (NCL-encoding) 6c**: Hier wurde die Kodierung von Tabelle 3 verwendet. Delays müssen Dank der Phaseneigenschaft (NULL vs. DATA) keine beachtet werden.

- **2-phase dual-rail (LEDR[3]-encoding) 6d**: Hier wurde die Kodierung aus Abbildung 4a verwendet. Delays müssen Dank der Phaseneigenschaft (φ_0 vs. φ_1) keine beachtet werden.

[2]inkl. Setuptime von den Latches
[3]Level Encoded Dual Rail

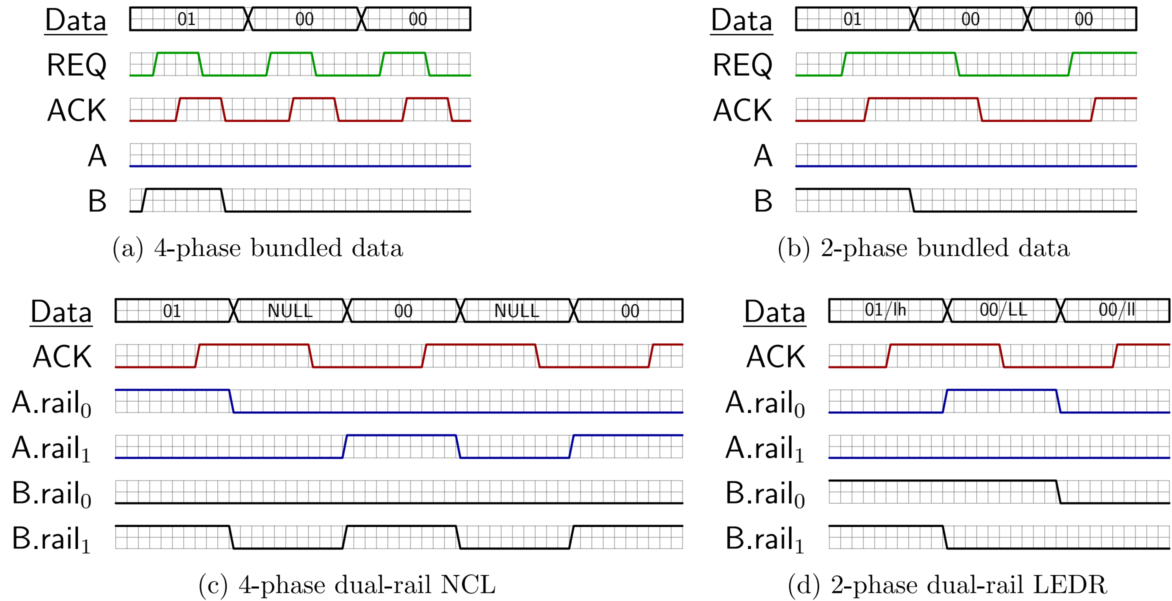


Abbildung 6: Timingdiagramme der verschiedenen Protokolle

Literatur

[SEE96] M. Shams, J. Ebergen, and M. Elmasry. A comparison of cmos implementations of an asynchronous circuits primitive: the c-element. In *Proceedings of the 1996 international symposium on Low power electronics and design*, ISLPED '96, pages 93–96, Piscataway, NJ, USA, 1996. IEEE Press.

[Sut89] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32:720–738, June 1989.