Load and Shiftregister: Schieberegister mit paralleler Lademöglichkeit der Flipflops

Assignment zum Modul:

Hardware-Design (EBS01)

07.10.2020

Vladimir Zhelezarov

.....

Studiengang: Digital Engineering und angewandte Informatik - Bachelor of Engineering

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis

Programmlistings

1 Einleitung

Digitale Schaltungen kommen in verschiedenen Varianten vor und dienen unterschiedlichen Zwecken. Es ist nicht immer eine direkte Verarbeitung der anliegenden Daten erforderlich, sondern auch deren Speicherung oder auch Transport durch verschiedene Mittel. Die Anforderungen orientieren sich individuell nach dem konkreten Einsatz und dadurch ist eine Analyse der Gegebenheiten, bevor die eine oder die andere Schaltung ausgewählt werden kann, notwendig. Um Kosten und Aufwand zu sparen, ist es wünschenswert, digitale Designs wieder zu verwenden. Dabei ist ein Kompromiss zwischen Komplexität und Vielseitigkeit der Schaltung einzugehen.

In dieser Arbeit wird ein flexibles Design auf der Basis eines Schieberegisters vorgestellt, welches für serielle und parallele Verarbeitung oder Speicherung von Daten benutzt werden kann. Die Entwicklung und Beschreibung des Designs erfolgt in der Hardwarebeschreibungssprache VHDL.

Um einen gemeinsamen Punkt der weiteren Betrachtungen im Kontext der weiteren Entwicklung zu sichern, müssen zuerst Begriffe und Definitionen geklärt werden. Für die Erstellung des Registers zeigen sich zwei Grundbausteine als notwendig - das Flipflop und der Multiplexer. Bei den Flipflops sind verschiedene Varianten denkbar und die Auswahl einer muss begründet werden.

Mit geklärte Spezifikationen erfolgt die Entwicklung in VHDL - zuerst von den einzelnen Bausteine und dann vom kompletten Register. Die Bausteine werden als Baublöcke oder in der VHDL-Terminologie - Komponenten betrachtet. Eine Betonung bekommt dabei die Flexibilität der Struktur und die Einstellung der Wortbreite durch Parameter.

Auch wenn der Code kompiliert werden kann, heißt es nicht unbedingt, dass die richtige Funktion gegeben ist. Dafür muss ein Szenario gefunden werden, in dem alle denkbare Interaktionen mit der entwickelten Schaltung vorhanden sind. Das Verhalten der Schaltung muss auch in geeigneter Weise kontrolliert werden.

2 Grundlagen

2.1 Begriffe

In digitalen Schaltungen werden Informationen als diskrete Werte dargestellt und verarbeitet[1]. Das hat den Vorteil, dass sich diese Schaltungen mit der Theorie der booleschen Algebra beschreiben lassen. Die mathematische Modelle dieser Algebra bieten mächtige Werkzeuge für die Beschreibung und Analyse des Verhaltens von solchen Schaltungen an[2]. Die Grundbausteine der digitalen Schaltungen sind die Logikgatter wie AND, OR, NOT, XOR sowie auch Kombinationen davon. Dadurch lassen sich auch komplizierte Modelle aufbauen. Wenn Werte aus dem Ausgang der Schaltung zurück zu ihrem Eingang geführt werden, entstehen Gedächtnis-Elemente. Somit ist die Ausgabe nicht mehr nur von den Eingangswerten abhängig, sondern auch von dem letzten internen Zustand. Die so entwickelte Schaltung wird als sequenziell bezeichnet[3]. Wenn keine Speicherelemente und dadurch Gedächtnis-Elemente vorhanden sind, ist die Schaltung kombinatorisch.

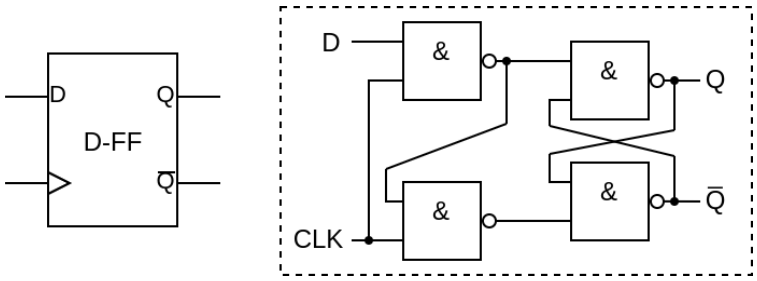
Für die Beschreibung digitaler Schaltungen kommen neben Schaltpläne auch die so genannten Hardwarebeschreibungssprachen in Frage. Die in dieser Arbeit benutze Sprache - VHDL - eignet sich hervorragend für die Darstellung der verschiedenen funktionellen, strukturellen und physikalischen Eigenschaften eines jedes digitalen Systems[4].

Digitale Schaltungen können auch miteinander kommunizieren. Dabei ist diese Kommunikation parallel, wenn mehrere Leitungen die Bits gleichzeitig übertragen, oder seriell, wenn die Information auf einer Leitung Bit für Bit transferiert wird[5].

2.2 Flipflop

Der Grundbaustein für die sequenzielle Schaltungen ist das Flipflop[6]. In seiner Grundform hat dieses Teil zwei Eingänge - für Setzen und Rücksetzen. Eine Aufbaumöglichkeit besitzt nur ein Set-Signal, wessen Ablesung vom CLK-Signal gesteuert wird. Damit wird die Schaltung zum D-Flipflop. Die folgende Abbildung zeigt den Aufbau:

[1] Vgl. Horowitz; Hill (2017), S.703
[2] Vgl. Fricke (2018), S.15
[3] Vgl. Gehrke et al. (2016), S.115
[4] Vgl. Ashenden (2008), S.7
[5] Vgl. Horowitz; Hill (2017), S.1027
[6] Vgl. Horowitz; Hill (2017), S.709



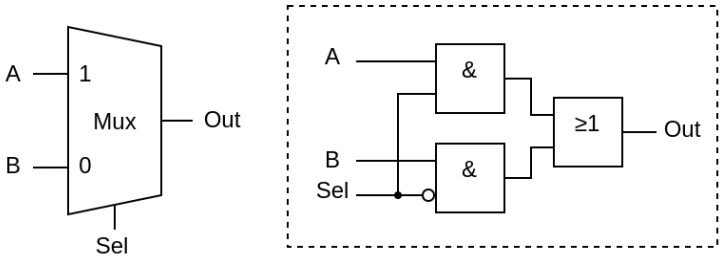
D	CLK	Q	$\bar{Q}$
0	0	unv.	unv.
1	0	unv.	unv.
0	1	0	1
1	1	1	0

Abbildung 1: D-Flipflop: Schaltzeichen, Grundaufbau und Wahrheitstabelle

Diese einfachste Form des D-Flipflops hat Probleme wie z.B. die ständige Abfrage des D-Eingangs, während CLK hoch ist. Zusätzlich fehlt die Möglichkeit den Ausgang direkt anzusteuern um einen definierten Zustand z.B. am Start zu sichern. Das erste Problem umgeht das D-Flipflop durch Gatter-Erweiterungen, die dazu führen, dass der Eingang nur bei der steigenden Flanke des CLK-Signals abgelesen wird. Das zweite Problem wird durch Zufügen von einem dominanten asynchronen Reset-Signal gelöst. Der Aufbau eines typischen industriell gefertigten D-Flipflops von Texas Instruments[1] mit solchen Eigenschaften ist im Anhang dargestellt und simuliert[2].

2.3 Multiplexer

Ein Multiplexer in seiner einfachsten Form ist ein Baustein, der einen von zwei digitale Eingänge auf seinen Ausgang schaltet. Der Eingang wird durch eine Select-Leitung ausgewählt[3].



Sel	A	B	Out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Abbildung 2: Multiplexer: Schaltzeichen, Grundaufbau und Wahrheitstabelle

[1] Nach Texas Instruments Inc. (1988), S.1
[2] auf Seite v
[3] Vgl. Fricke (2018), S.111

2.4 Schieberegister

Wenn mehrere Flipflops miteinander verbunden sind, sodass jeder Q-Ausgang den nächsten D-Eingang schaltet, ist die so erzeugte sequenzielle Schaltung ein Schieberegister[1]. Durch Zufügen von Multiplexer zwischen den Flipflops, Ablesen der einzelnen Flipflop-Ausgänge und eventuelle Rückkoppelungen entstehen die unterschiedlichen Varianten von Schieberegister. Sie werden klassifiziert nach dem Format der Daten, die am Ein- und Ausgang des Schieberegisters anliegen: Seriell-Ein/Seriell-Aus, Parallel-Ein/Parallel-Aus, Seriell-Ein/Parallel-Aus, Universell und der Ringzähler[2].

Für die Ziele dieser Arbeit definieren wir ein Load and Shiftregister oder kurz LSRegister als ein Schieberegister, das über serielle und parallele Ein- und Ausgänge verfügt. Die Ladung der Flipflops wird über Multiplexer zwischen seriell und parallel gewählt.

Aufgrund der Übersichtlichkeit betrachten wir das Prinzip eines 4-Bit LSRegisters. Längere Wortbreiten folgen denselben Regeln.

Der serielle Eingang ist Din, serieller Ausgang ist Dout, die vier parallelen Eingänge sind $[D_0 .. D_4]$ und die vier parallelen Ausgänge - $[Q_0 .. Q_4]$

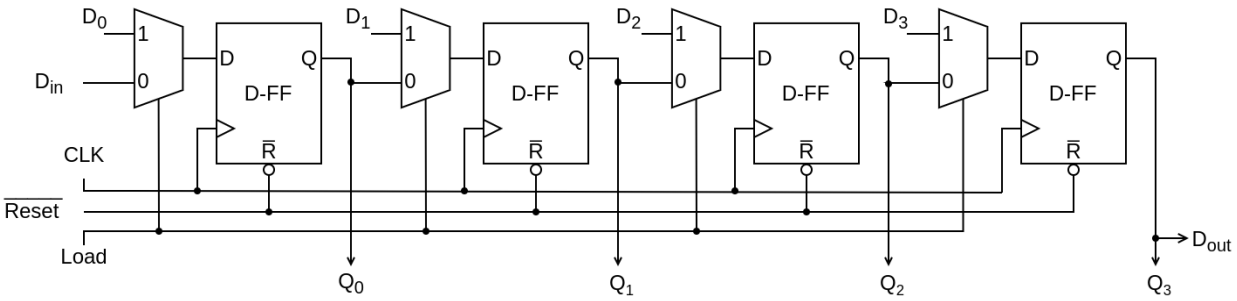


Abbildung 3: 4-Bit LSRegister

Die folgenden Gleichungen beschreiben das Verhalten dieser Schaltung:

$$Q_i^m = 0 \quad \text{für Reset=0} \tag{1}$$

$$Q_0^{m+1} = (\overline{Load} \wedge D_{in}) \vee (Load \wedge D_0) \quad \text{für 0<i≤4, Reset=1} \tag{2}$$

$$Q_i^{m+1} = (\overline{Load} \wedge Q_{i-1}^m) \vee (Load \wedge D_i) \quad \text{für 0<i≤4, Reset=1} \tag{3}$$

$$D_{out} = Q_3^m \tag{4}$$

[1] Vgl. Horowitz; Hill (2017), S.744
[2] Kuphaldt (2007), S.350

3 Entwicklung in VHDL

Es bietet sich an, die Komponenten der entwickelten Schaltung in separaten Dateien unterzubringen, um einen besseren Überblick über die Code-Struktur zu sichern. Das LSRegister kann diese nach Bedarf in den benötigten Zahlen generieren, eingestellt durch ein *Generic*-Parameter[1]. Bei geänderten Anforderungen an der Wortbreite der Schaltung ist nur der Parameter zu ändern. Dieses Mechanismus trägt zu einer flexiblen Code-Struktur bei[2].

Die erforderliche Bausteine für das LSRegister sind das D-Flipflop und der Multiplexer.

3.1 Baustein D Flipflop

Bei der Beschreibung in VHDL definieren wir als Eingänge neben dem seriellen Eingang auch das Reset- und das Uhr-Signal, da diese auch das Verhalten der Schaltung beeinflussen. Der Ausgang besteht nur aus dem Ausgabewert q vom Flipflop.

Wichtig für die Funktion ist die Uhr-unabhängige Funktion vom Reset-Signal. Dafür landet das Signal zusammen mit der Uhr in der Sensitivitätliste des Prozesses. Das führt zu einer sofortigen Reaktion der Schaltung bei Änderung von jedem der beiden Werte[3].

Listing 1: D Flip-Flop mit inv. Reset

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY d_ff IS
5  PORT( d, clk, rb: IN std_logic;
6        q: OUT std_logic);
7  END d_ff;
8
9  ARCHITECTURE behave OF d_ff IS
10 BEGIN
11    PROCESS (clk, rb)
12    BEGIN
13       IF rb = '0' THEN
14          q <= '0';
15       ELSIF rising_edge(clk) THEN
16          q <= d;
17        END IF;
18    END PROCESS;
19 END ARCHITECTURE behave;
```

Die nachfolgende Simulation mit dem Simulationswerkzeug ModelSim[4] zeigt die korrekte

[1] The Institute of Electrical and Electronics Engineers, Inc. (1993), S.6-7
[2] Vgl. Ashenden (2008), S.365
[3] Vgl. The Institute of Electrical and Electronics Engineers, Inc. (1993), S.127
[4] Altera Corporation (2013), Kap.2, S.9

Funktion der Schaltung:

- Die Schaltung reagiert auf die steigende Flanke der Uhr;
- Bei der Abfrage vom Eingang wird sein Wert am Ausgang übernommen;
- Der Wert am Ausgang bleibt erhalten bis zur nächsten Eingangabfrage;
- Ein hochgezogenes Reset-Signal führt sofort, unabhängig von der Uhr, zu einer Rücksetzung des Ausgangs auf Null.

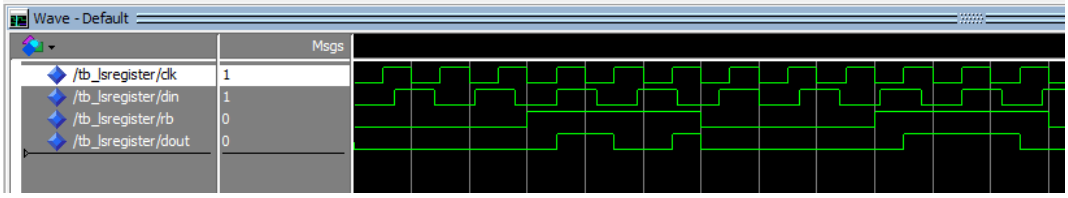


Abbildung 4: D-Flipflop: Simulation mit ModelSim

3.2 Baustein Multiplexer

Der Multiplexer ist vergleichbar einfach aufgebaut, indem die gesuchte Funktion durch eine logische Verknüpfung realisiert wird. Der Ausgang ist entweder gleich dem einen Eingang oder dem anderen, je nach Wert des *Select*-Signals.

Listing 2: 2-Weg Multiplexer

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY mux IS
5  PORT( a, b, sel: IN std_logic;
6        q: OUT std_logic);
7  END mux;
8
9  ARCHITECTURE behave OF mux IS
10 BEGIN
11    q <= (a AND sel) OR (b AND (NOT sel));
12 END ARCHITECTURE behave;
```

Bei der Simulation sind als Eingänge zwei Takten mit verschiedenen Frequenzen ausgewählt, was für bessere Visualisierung beiträgt. Der Wert des Ausgangs bildet beide Signale nacheinander ab, je nach Einstellung mit *Select:*

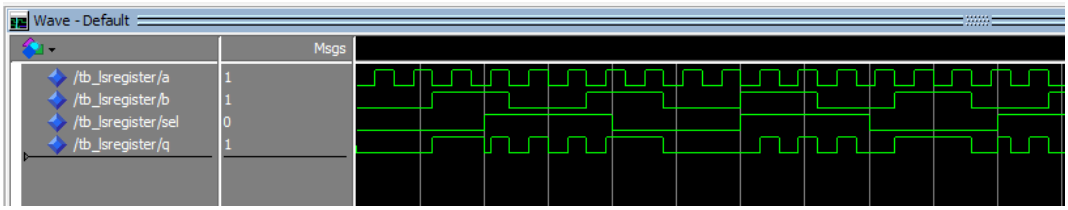


Abbildung 5: Multiplexer: Simulation mit ModelSim

3.3 Aufbau des LSRegisters

Als eine Stufe des LSRegisters betrachten wir ein D-Flipflop mit Reset-Signal, wessen Eingang durch einen Multiplexer geschaltet wird, wie in der folgenden Abbildung dargestellt:

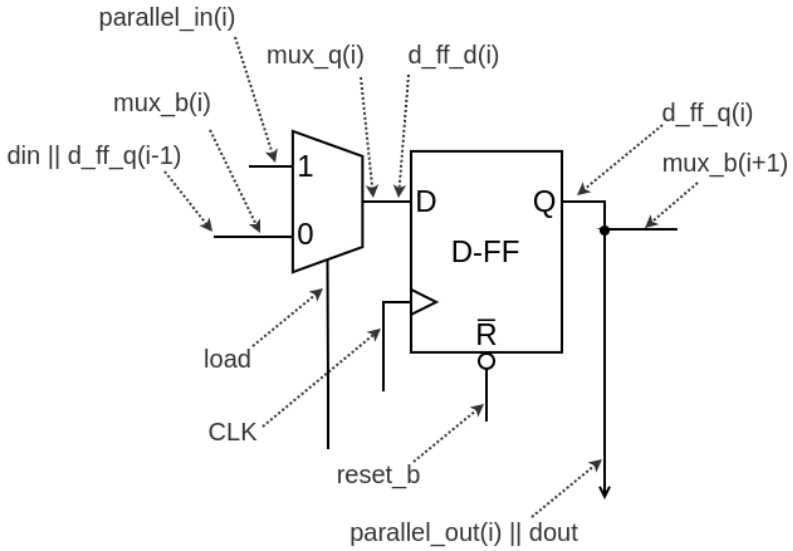


Abbildung 6: VHDL Signalzuweisungen

Eingetragen in der Abbildung sind auch die Signale, die für die VHDL-Beschreibung notwendig sind. Beide Bausteine - das Flipflop und der Multiplexer werden als *COMPONENT*[1] initialisiert und an den in der Abbildung gezeigten Signale durch *PORT MAP*[2] gebunden.

Spezielle Betrachtung braucht nur die erste Stufe, weil dabei der Eingang des Multiplexers auch der Eingang der Schaltung ist. Der Rest kann nach gleichen Regeln generiert werden, indem jeder 0-Eingang der Multiplexer zu dem Ausgang des vorigen Flipflops verbunden ist.

Als serielle Eingänge sind der Din-Eingang und die gemeinsamen Leitungen für Load, CLK und Reset geschaltet. Die parallelen Eingänge sind direkt an der 1-Eingänge der Multiplexer gebunden und die parallelen Ausgänge werden an dem Q-Ausgang jedes Flipflops abgelesen.

[1]The Institute of Electrical and Electronics Engineers, Inc. (1993), S.134
[2]The Institute of Electrical and Electronics Engineers, Inc. (1993), S.77

Der komplette VHDL-Code für das nach Wortbreite konfigurierbares LSRegister befindet sich im Anhang[1].

4 Verifikation und Anwendungen

4.1 Simulation einer Seriell-parallele Kommunikation

Ein mögliches Simulationsszenario bei dem alle Ein- und Ausgänge der generierten Schaltung benutzt werden, ist eine seriell-parallele Kommunikation zwischen zwei LSRegister. Als ein Standartwert für die Wortbreite ist 8-Bit eingestellt. Die Simulation benutzt die Benennung *DUT* - Design-Under-Test für beide simulierte Schaltungen. Abbildung 7 zeigt den Aufbau.

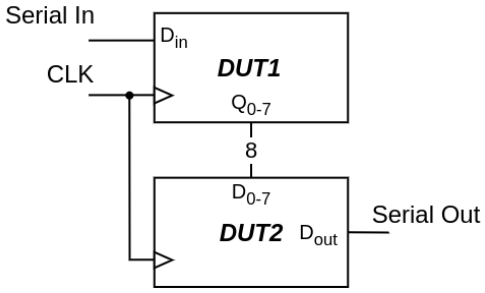


Abbildung 7: 8-Bit Serielle Kommunikation: Aufbau

Das erste Modul *DUT1* bekommt die Daten seriell an seinem *Din*-Eingang und überträgt diese parallel am zweiten Modul *DUT2*. Dieses stellt die Daten wieder seriell um und gibt sie an seinem seriellen Ausgang *Dout* aus. Beide Module teilen eine gemeinsame Uhr, damit sie synchronisiert arbeiten. Alle Elemente müssen zum Anfang resettet werden. Somit ist sichergestellt, dass keine undefinierte Zustände bei den Ein- oder Ausgängen vorhanden sind. *DUT1* muss nach dem Laden eines ganzen Wortes ein Uhr-Zyklus warten, weil diese Zeit von *DUT2* benutzt wird um das Wort parallel zu lesen. Nachdem das Wort in *DUT2* geladen ist, schaltet das Modul auf Schiebemodus um. Wenn das Wort seriell zum Ausgang komplett geschoben wird, ist schon das nächste Wort vom *DUT1* geladen und kann übernommen werden. Der ganze Algorithmus wiederholt sich solange die Zeitdauer der Simulation nicht eine vordefinierte Dauer überschreitet. Zum Abfragen der vergangenen Simulationszeit bietet VHDL die Funktion *NOW* an[2].

Zur Verifikation der Ergebnisse wird ein paralleler Prozess mit *Assert*-Anweisungen eingesetzt. Diese prüfen das boolesche Ergebnis einer Aussage und melden oder sogar unterbrechen die Simulation bei deren Nichteinhaltung[3]. Weil beide Module zeitlich versetzt arbeiten,

[1] auf Seite vi
[2] The Institute of Electrical and Electronics Engineers, Inc. (1993), S.200
[3] Vgl. Ashenden (2008), S.87-88

kann *DUT1/Din* nicht einfach mit *DUT2/Dout* verglichen werden. Der Prüfprozess nutzt dafür den Fakt aus, dass das selbe Testwort immer wieder übertragen wird und vergleicht es bitweise mit dem *Dout* von *DUT2*. Wenn diese übereinstimmen, ist die richtige Funktion vorhanden. Der Prüfprozess orientiert sich zeitlich nach den selben Abständen, wie beide Module - Wort lesen/schreiben, ein Zyklus Pause einlegen. Zusätzlich berücksichtigt er eine Wartezeit beim Start der Schaltung, die für das Resetten benötigt wird. Weil die Prüfung, ob die Simulation zum Ende gekommen ist, nur vor jedem Wort erfolgt, kann dabei passieren, dass ein Modul früher gestoppt hat, als der Prüfprozess. Dann wird dieser falsche Fehlermeldungen anzeigen. Dafür wird der Prüfprozess ein Wort früher gestoppt.

Pseudocode für die Algorithmen ist in den folgende Listings dargestellt. Aufgrund seiner Einfachheit und der starken Ähnlichkeit zu *DUT1* und *DUT2* wird auf eine Darstellung mit Pseudo-Code für den Prüfprozess verzichtet. Der VHDL-Code für die komplette Testbench befindet sich im Anhang[1].

Listing 3: Pseudocode DUT1 (Seriell-zu-Parallel)

```
1  reset_circuit;
2  setmux(serial);
3  while elapsed_time < sim_length do
4      for i in 0 to word_length
5          read(serial);
6          shift;
7      end for;
8      wait(clock_cycle);
9  end while;
```

Listing 4: Pseudocode DUT2 (Parallel-zu-Seriell)

```
1  reset_circuit;
2  setmux(parallel);
3  wait(word_length*clock_cycle);
4  while elapsed_time < sim_length do
5      read(parallel);
6      setmux(serial);
7      for i in 0 to word_length
8          shift;
9      end for;
10     setmux(parallel);
11 end while;
```

Eine grafische Darstellung der Ergebnisse bietet zusätzliche Hilfe bei der Verifizierung der richtigen Funktion der Schaltung an. Aufgrund von Übersichtlichkeit folgen Abbildun-

[1] auf Seite vii

gen von den Zeitpunkten, die wichtig für die Synchronisation sind und bei den die interne parallele Übertragung stattfindet. Der Rest der Funktion besteht bei beiden Modulen aus einer einfachen seriellen Verschiebung nach rechts.

Beide parallele Leitungen sind direkt miteinander verbunden und dafür sind auch ihre Werte gleich. Bedeutend für die Synchronisation und dadurch für die richtige Funktion ist die Umschaltung von *dut2_load* - das Steuersignal vom Multiplexer bei *DUT2*. Dabei müssen beide Module ein Zyklus abwarten, damit die Bits in *DUT2* übertragen werden. Danach können die Register die Schiebung fortsetzen.

Auf Abbildung 8 ist das Testwort „*10111001*“ zum Zeitpunkt 90ns komplett in *DUT1* geladen und liegt an den parallelen Ausgängen von *DUT1* bzw. an den Eingängen von *DUT2* an. Bei der nächsten Flanke um 110ns ist der Multiplexer schon umgestellt und das Wort wird von den Flipflops gelesen.

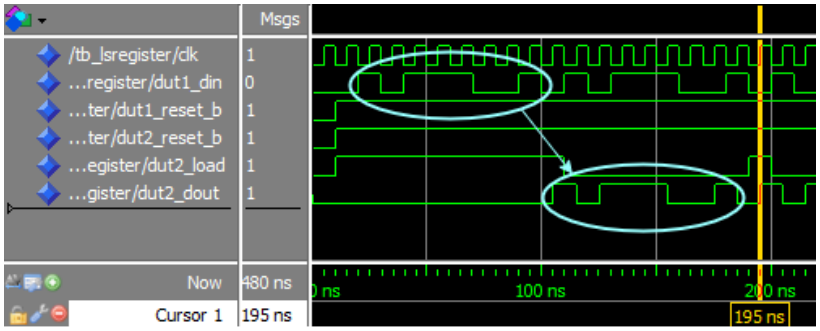


Abbildung 8: 8-Bit Serielle Kommunikation: Simulation mit ModelSim

Auch wenn die parallelen Ausgänge von *DUT2* nicht für die Kommunikation benötigt werden, liegen dort immer die Werte der Flipflop-Ausgänge an und können in der Simulation abgefragt werden. Abbildung 9 zeigt einen Zyklus weiter, wenn bei der Taktflanke um 195ns und hochgezogenes *dut2_load* das Wort übernommen wird.

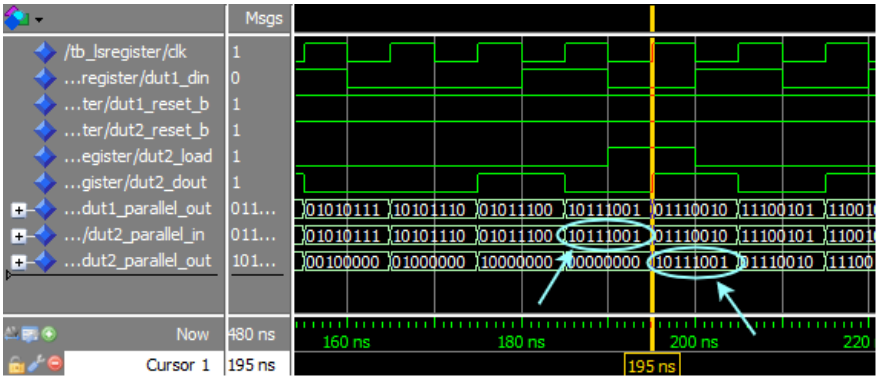


Abbildung 9: 8-Bit Serielle Kommunikation - nächster Zyklus

4.2 Anwendungen des LSRegisters

Die vorgestellte Schaltung, bedingt durch ihren zahlreichen Ein- und Ausgänge kann Anwendung in viele Szenarien finden. Ohne Anspruch auf Vollständigkeit sind diese:

Als Wandler von seriell auf parallel (Seriell-Ein/Parallel-Aus oder umgekehrt):

Wenn die vorhandenen Ports eines Mikrocontrollers nicht reichen, kann ein seriell-paralleler Wandler neue Ein-/Ausgänge anbieten. Andersherum, wenn der Mikrocontroller ein Bauteil mit parallelen Eingänge steuern möchte, müssen nicht unbedingt so viele Ports vom Controller besetzt werden, vor allem nicht, wenn die Kommunikation keine schnelle Übertragung benötigt. Auch bei paralleler Kommunikation können mit zwei solchen Wandler Leitungen und den damit verbundenen Aufwand gespart werden.

Als serielles Schieberegister (Seriell-Ein/Seriell-Aus):

Aus den boolesche Eigenschaften der Schaltung folgt, dass eine Schiebung nach rechts eine Division durch zwei entspricht. Umgekehrt ist die linke Schiebung eine Multiplikation mit zwei. Wenn diese Operation wiederholt wird, dient das serielle Schieberegister als Multiplikations- oder Divisionsschaltung für alle 2-er Potenzen, bis auf seiner Länge.

Es kann unter Umständen notwendig sein, dass ein Signal verzögert zu der nächsten Verarbeitungseinheit weitergeleitet werden muss. In diesem Fall entspricht jede Stufe des seriellen Schieberegisters eine Verzögerung um einen Taktzyklus.

Bei der Ablesung eines Tasters kann durchaus vorkommen, dass der Wert nicht schnell und einmalig wechselt, sondern sporadisch springt. In dieser Situation kann das LSRegister, geschaltet als serieller Schieberegister als Entprellung-Schaltung dienen. Die Anzahl der Stufen wird die Dauer bestimmen, für welche das abgelesene Signal stabil anliegen muss, um anerkannt zu werden.

Wenn einige der parallelen Ausgänge über XOR-Verknüpfungen zurück zum Eingang gekoppelt werden, entsteht ein lineares rückgekoppeltes Schieberegister[1]. Diese Schaltung wird unter anderem als Zufallszahlengenerator benutzt.

Als Speicherelement (Parallel-Ein/Parallel-Aus):

Geschaltet als Parallel-Ein/Parallel-Aus kann das LSRegister als Speicherelement dienen - ganze Speicherwörter werden auf den Flipflops gespeichert und liegen an den parallelen Ausgänge an. Eventuell ist dabei eine Erweiterung um ein Enable-Signal notwendig, das über AND-Gatter das CLK-Signal schaltet.

Als FIFO (Eimerkettenspeicher):

Wenn ein Teilnehmer am Eingang der Schaltung seriell schreibt und der andere Teilnehmer die Daten am Ausgang seriell abliest, entsteht ein FIFO (First in First out) Puffer[2]. Somit sind beide zu einem gewissen Maß entkoppelt und arbeiten asynchron.

[1] Vgl. Gehrke et al. (2016), S.183
[2] Vgl. Fricke (2018), S.174

Weitere Anwendungen:

Durch eine dauerhafte Ansteuerung des Load-Signals auf logische Eins besteht die Möglichkeit weitere Elemente zwischen den Flipflops einzusetzen, indem diese zwischen Q_i und D_i zwischengeschaltet werden. Dadurch ist die Flexibilität der Schaltung noch größer.

5 Zusammenfassung

Schieberegister kommen in verschiedenen Varianten und Formen vor und haben zahlreiche Anwendungen in vielen Bereiche der digitalen Technik. Dabei ist eine Balance zwischen der Komplexität der Schaltung und ihrer Vielseitigkeit zu finden. Als eine Form von universelles Schieberegister wurde in dieser Arbeit ein Design vorgestellt, das seriell und parallel arbeiten kann und dadurch eine starke Wiederverwendbarkeit genießt. Das Konzept basiert auf Flipflops, deren Eingänge über Multiplexer geschaltet werden. Um eine flexible Struktur zu erreichen ist die Wortbreite der Schaltung per Parameter angegeben und alle Bausteine sind als Komponenten angebunden.

Die richtige Funktion zeigt eine Simulation mit dem Werkzeug ModelSim, bei der zwei solche Register miteinander parallel kommunizieren. Dabei wird das Testwort seriell im ersten Modul eingespeist und auch seriell aus dem zweiten Modul abgelesen. Zur Verifikation wurde neben einer grafischen Darstellung der Signale auch eine Kontrolle der Einzelbits über Assert-Anweisungen benutzt.

Dank der vielen Ein- und Ausgänge findet das entwickelte Register Anwendung in mehrere Szenarien. Je nach Benutzung und Anschluss kann es als seriell-paralleler Wandler, Schieberegister, Entprellung, Speicher und andere dienen. Bei geeigneter Verbindung der Pins, eröffnet sich die Möglichkeit, dass extra Gatter zwischen den Flipflops geschaltet werden können. Damit steigt die Anzahl an Einsatzmöglichkeiten noch mehr. Diese extra Anwendungen wurden allerdings nicht mit Testbenches getestet.

Bei dem Entwurf sequenzieller Schaltungen werden oft endliche Automaten als mathematisches Modell benutzt. Weil eine gute Übersichtlichkeit der Funktion bei der betrachtete Schaltung durch Diagramme, Code und Beschreibung gegeben ist, sowie auch aus Platzgründen, wurde in der Arbeit auf eine Modellierung mit endliche Automaten verzichtet.

Eine Installation und Inbetriebnahme auf Hardware wurden nicht durchgeführt. Die dabei verbundenen Artefakte und Tätigkeiten wie Pin-Assignments, Hardware-Optimierung oder Analyse der Synthese-Berichte wurden auch erspart.

Die Simulation kann nicht auf hundert Prozent garantieren, dass die reale Hardware sich auch so verhält, wie erwartet. Es sind Fehlkonditionen denkbar, oder auch Zeitverzögerungen die in der Simulation nicht berücksichtigt werden. Bei der Arbeit mit konkreter Hardware spielen auch ihre spezifischen Eigenschaften eine große Rolle bei der Optimierung des Designs.

Es wurde implizit angenommen, dass die Signale sich mit Null-Verzögerung durch die Schaltung bewegen, was nicht realistisch ist. Für weitere und präzisere Betrachtungen ist die Arbeit mit echter Hardware notwendig.

Literaturverzeichnis

Altera Corporation (2013)
Quartus II Handbook Version 13.0, Volume 1: Design and Synthesis, o.O.

Ashenden, P. J. (2008)
The designer's guide to VHDL, Third Edition, Amsterdam et al.

Fricke, K. (2018)
Digitaltechnik: Lehr-und Übungsbuch für Elektrotechniker und Informatiker, 8., überarbeitete und aktualisierte Auflage, Wiesbaden

Gehrke, W. et al. (2016)
Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller, 7., überarbeitete und aktualisierte Auflage, Berlin

Horowitz, P.; Hill, W. (2017)
The art of electronics, Third Edition, New York

Kuphaldt, T. (2007)
Lessons In Electric Circuits, Volume IV–Digital, Fourth Edition, o.O.

Texas Instruments Inc. (1988)
Dual D-Type Positive-Edge-Triggered Flip-Flops With Preset and Clear, https://www.ti.com/lit/ds/symlink/sn74ls74a.pdf (Zugriff am 07.10.2020)

The Institute of Electrical and Electronics Engineers, Inc. (1993)
IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-1993, New York

Anhang

Anhang 1: Digitale Schaltungen

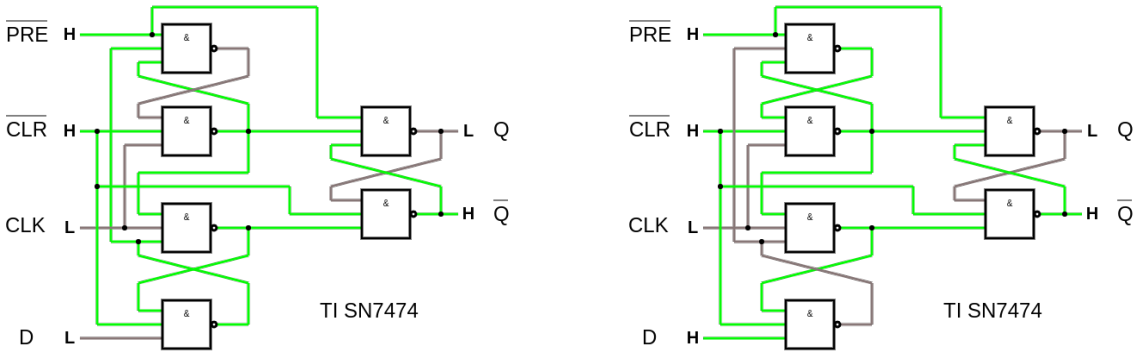


Abbildung 10: TI SN7474 Simulation 1/2

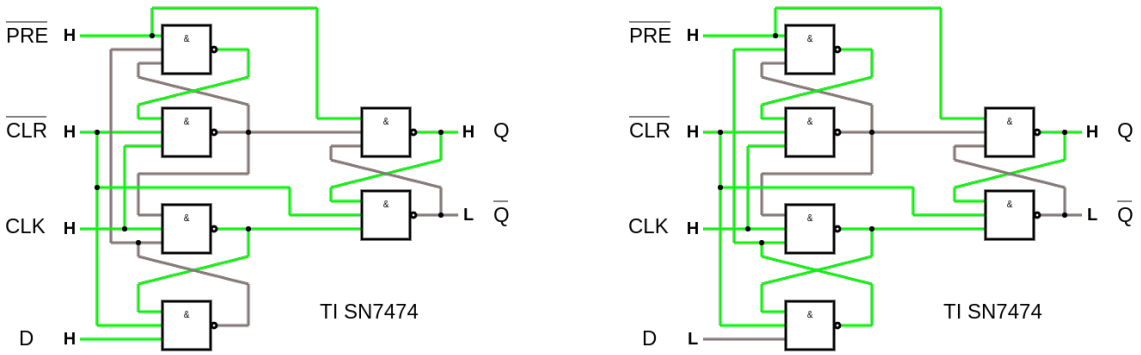


Abbildung 11: TI SN7474 Simulation 2/2

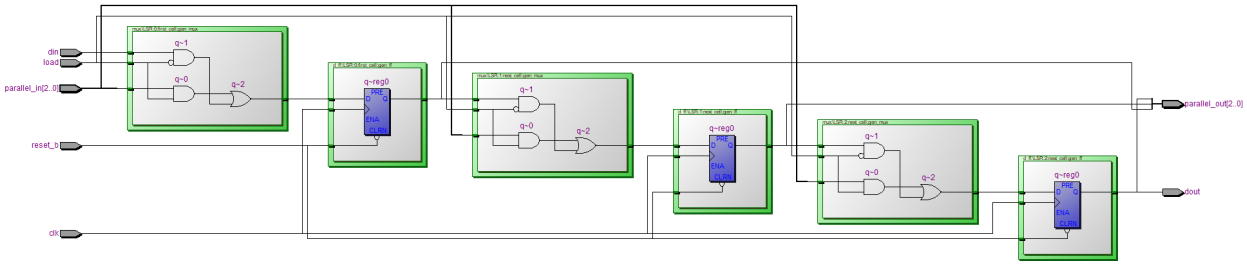


Abbildung 12: 3-Bit LSRegister: RTL-Sicht

Anhang 2: VHDL Code

Anhang 2.1: Load and Shift Register

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY LSRegister IS
5     GENERIC (lsr_width: positive := 3);
6     PORT( din: IN std_logic;
7           dout: OUT std_logic;
8           clk, load, reset_b: IN std_logic;
9           parallel_in: IN std_logic_vector((lsr_width-1) DOWNTO 0);
10          parallel_out: OUT std_logic_vector((lsr_width-1) DOWNTO 0) );
11 END ENTITY LSRegister;
12
13 ARCHITECTURE behave OF LSRegister IS
14
15    -- D Flipflop
16    COMPONENT d_ff IS
17       PORT( d, clk, rb: IN std_logic;
18             q: OUT std_logic);
19    END COMPONENT d_ff;
20
21    -- 2-way multiplexer
22    COMPONENT mux IS
23       PORT( a, b, sel: IN std_logic;
24             q: OUT std_logic);
25    END COMPONENT mux;
26
27    CONSTANT last_index: natural := lsr_width-1;
28
29    -- internal signals
30    SIGNAL mux_b, mux_q, d_ff_d, d_ff_q: std_logic_vector(last_index DOWNTO 0);
31
32 BEGIN
33    -- serial input/output
34    mux_b(0) <= din;
35    dout <= d_ff_q(last_index);
36
37    -- generate cells
38    LSR:
39    FOR i IN 0 TO last_index GENERATE
40
41       gen_mux: COMPONENT mux PORT MAP(a => parallel_in(i), b => mux_b(i),
42                                       sel => load, q => mux_q(i));
43       gen_ff: COMPONENT d_ff PORT MAP(d => mux_q(i), clk => clk,
44                                       rb => reset_b, q => d_ff_q(i));
45       parallel_out(i) <= d_ff_q(i);
46
47       next_cell: IF i > 0 GENERATE
48          mux_b(i) <= d_ff_q(i-1);
49       END GENERATE next_cell;
50
```

```
51      END GENERATE LSR;
52
53  END ARCHITECTURE behave;
```

Anhang 2.2: Testbench für 8-Bit seriell-parallele Kommunikation

---

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3
4   ENTITY tb_LSRegister IS
5   END ENTITY tb_LSRegister;
6
7   ----------------------------
8   -- Serial communication test
9   ----------------------------
10  ARCHITECTURE behave_lsr OF tb_LSRegister IS
11      CONSTANT test_width:    positive:= 8;
12      CONSTANT simlength:    time:= 1 us;
13      CONSTANT test_word:    std_logic_vector:="10111001";
14
15      CONSTANT last_index:    natural:= test_width-1;
16      SUBTYPE  lsrvector IS    std_logic_vector(last_index DOWNTO 0);
17
18      -- single "tic" of the clock
19      CONSTANT clocktime:     time:= 10 ns;
20      CONSTANT word_duration:   time:= test_width*clocktime;
21
22      -- constants to compare to in the asserts
23      CONSTANT zeros:      lsrvector:= (others => '0');
24
25      SIGNAL  clk,
26                dut1_din,
27                dut1_dout,
28                dut1_reset_b,
29                dut1_load,
30                dut2_din,
31                dut2_dout,
32                dut2_reset_b,
33                dut2_load:     std_logic:='0';
34      SIGNAL  dut1_parallel_in,
35                dut1_parallel_out,
36                dut2_parallel_in,
37                dut2_parallel_out:  lsrvector:= (others => '0');
38  BEGIN
39      dut1: ENTITY work.LSRegister(behave)
40          GENERIC MAP(lsr_width => test_width)
41          PORT MAP( dut1_din,   dut1_dout,    clk,  dut1_load,
42                        dut1_reset_b,  dut1_parallel_in, dut1_parallel_out);
43
44      dut2: ENTITY work.LSRegister(behave)
45          GENERIC MAP(lsr_width => test_width)
```

```
46          PORT MAP( dut2_din,   dut2_dout,    clk,  dut2_load,
47                       dut2_reset_b,  dut2_parallel_in, dut2_parallel_out);
48
49      -- simulate the parallel cable connection
50      dut2_parallel_in <= dut1_parallel_out;
51
52      -- a global clock
53      clock: PROCESS
54      BEGIN
55          IF now < simlength THEN
56              WAIT FOR clocktime/2;
57              clk <= NOT clk;
58          ELSE WAIT;
59          END IF;
60      END PROCESS clock;
61
62      -- dut1: serial to parallel
63      SIPO: PROCESS
64      BEGIN
65          -- give time to reset
66          WAIT FOR clocktime;
67          -- ready for shifting
68          dut1_reset_b  <= '1';
69          dut1_load   <= '0';
70          WAIT FOR clocktime;
71          WHILE now < simlength LOOP
72              -- load the test word
73              FOR i IN 0 TO last_index LOOP
74                  dut1_din <= test_word(i);
75                  WAIT FOR clocktime;
76              END LOOP;
77              -- let dut2 read the test word
78              dut1_din <= '0';
79              WAIT FOR clocktime;
80          END LOOP;
81          WAIT;
82      END PROCESS SIPO;
83
84      -- dut2: parallel to serial
85      PISO: PROCESS
86      BEGIN
87          -- give time to reset
88          WAIT FOR clocktime;
89          -- ready for parallel load
90          dut2_reset_b  <= '1';
91          dut2_load   <= '1';
92          WAIT FOR clocktime;
93          -- wait for the first word
94          WAIT FOR word_duration;
95          WHILE now < simlength LOOP
96              -- read the word
97              WAIT FOR clocktime;
98              dut2_load <= '0';
99              -- output the word
100             WAIT FOR word_duration;
101             dut2_load <= '1';
```

```
102         END LOOP;
103         WAIT;
104     END PROCESS PISO;
105
106     -- error checks
107     check: PROCESS
108     BEGIN
109         -- setup + load first word + offset
110         WAIT FOR (2*clocktime + word_duration - clocktime);
111         sim:
112         -- the checks should stop a whole word earlier
113         WHILE now < (simlength - word_duration) LOOP
114             WAIT FOR clocktime;
115             chkword:
116             FOR i IN 0 TO last_index LOOP
117                 WAIT FOR clocktime;
118                 ASSERT dut2_dout=test_word(i)
119                     REPORT  "Wrong bit at position "&integer'IMAGE(i)&
120                               ". It is "&std_logic'IMAGE(dut2_dout)&
121                               ", but should be:"&std_logic'IMAGE(test_word(i))&
122                               "." SEVERITY FAILURE;
123             END LOOP chkword;
124         END LOOP sim;
125         WAIT;
126     END PROCESS check;
127 END ARCHITECTURE behave_lsr;
```