

HARDWARE- BESCHREIBUNGSSPRACHEN

Hardwareentwurf mit VHDL

21. Oktober 2021
Revision: b941727 (2021-01-16 01:57:51 +0100)

Steffen Reith

Theoretische Informatik
Studienbereich Angewandte Informatik
Hochschule **RheinMain**

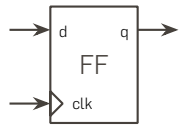


Notizen

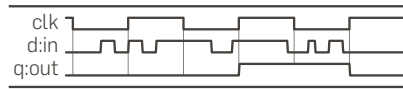
Notizen

ASYNCHRONE SIGNALE & CLOCK
DOMAIN CROSSING

FLIP-FLOPS / TIMING



Ein Flip-Flop ist **edge-sensitiv** und übernimmt Daten nur an (steigenden) **Flanken**.



In der Praxis¹ sind

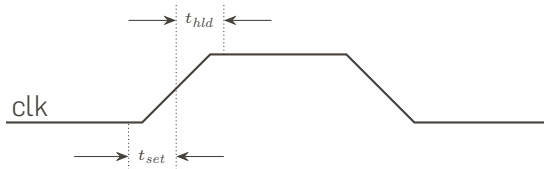
- Flanken **nicht senkrecht** und
- ein Flip-Flop **benötigt Zeit** um die Daten zu übernehmen.

Der Dateneingang darf sich für eine technologieabhängige Mindestzeit **vor** der Taktflanke nicht ändern (**Setup-Time** t_{set}) und muss auch noch **nach** der Taktflanke eine gewisse Zeit stabil sein (**Hold-Time** t_{hld}).

¹Literatur: Pong P. Chu, RTL Hardware Design Using VHDL, John Wiley & Sons, 2006

Notizen

METASTABILITÄT



Ändert sich der Dateneingang eines Flip-Flops von '0' nach '1' **innerhalb** der **Setup-Time** oder **Hold-Time**, so kann das Flip-Flop

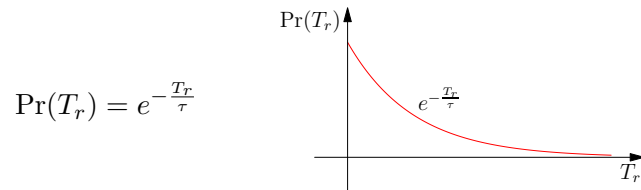
- '1' ausgeben (gewünscht),
- '0' ausgeben oder (alten Wert gespeichert, dann „klappt“ es das nächste Mal vielleicht)
- in einen **metastabilen Zustand** gelangen (der Spannungspegel befindet sich dann irgendwo zwischen '1' und '0'). Die **Ausgabe** kann **weder als '1'** noch als **'0'** interpretiert (nachfolgende Schaltkreise evtl. in undefiniertem Zustand).

Notizen

METASTABILITÄT (II)

Geringe Störungen (Rauschen) lassen das Flip-Flop wieder in einen **stabilen Zustand kippen**.

Die **Wahrscheinlichkeit**, dass sich das Flip-Flop nach der Zeit T_r („Resolution Time“) noch **im metastabilen Zustand** befindet beträgt:



(τ ist eine hardwareabhängige Konstante)

Bei heutigen Technologien beträgt τ Bruchteile von Nanosekunden.

Frage: Wie zuverlässig ist ein Schaltkreis, wenn **asynchrone** Eingaben benötigt werden?

179

Notizen

METASTABILITÄT (III)

Kann ein Flip-Flop den metastabilen Zustand nicht innerhalb eines gegebenen Zeitraums verlassen / beheben, dann spricht man von einem **Synchronisationsfehler**.

Die Zuverlässigkeit einer Schaltung beschreibt die **durchschnittliche Zeit zwischen zwei aufeinanderfolgenden Synchronisierungsfehlern** (kurz: $\text{MTBF}(T_r)$)

Seien ω das Zeitfenster in dem ein Flip-Flop in den **metastabilen** Zustand gerät (aktuell: einige Picosekunden bis Bruchteile von Nanosekunden), f_{clk} die **Taktfrequenz** und f_d die **Änderungsfrequenz** der Eingabe, dann gilt

$$\text{MTBF}(T_r) = \frac{e^{\frac{T_r}{\tau}}}{\omega \cdot f_{clk} \cdot f_d}$$

180

Notizen

MTBF - EIN BEISPIEL

Sei $\tau = 0.5\text{ns}$, $\omega = 0.1\text{ns}$, $f_{clk} = 50\text{Mhz}$ und $f_d = 0.1 \cdot f_{clk}$, dann

T_r	MTBF
0.0ns	$4.00 \cdot 10^{-5}\text{sec}$
2.5ns	$5.94 \cdot 10^{-3}\text{sec}$
5.0ns	$8.81 \cdot 10^{-1}\text{sec}$
10.0ns	$1.94 \cdot 10^4\text{sec}$ (5.39 Stunden)
20.0ns	$9.42 \cdot 10^{12}\text{sec}$ ($2.99 \cdot 10^5$ Jahre)
30.0ns	$4.57 \cdot 10^{21}\text{sec}$ ($1.45 \cdot 10^{14}$ Jahre)
35.0ns	$1.01 \cdot 10^{26}\text{sec}$ ($3.19 \cdot 10^{18}$ Jahre)

Entspricht T_r der halben Taktfrequenz, so ist die unakzeptable MTBF ca. 5 Stunden. Schon bei der 1.5fachen Resolution Time ergeben sich *ausreichende* 10^{14} Jahre (Alter des Universums: ca. 10^{11} Jahre).

181

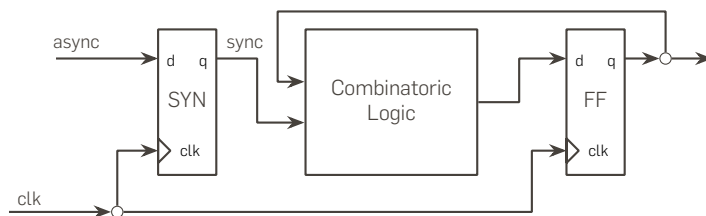
Notizen

UMGANG MIT SYNCHRONISATIONSFEBLERN

Synchronisationsfehler kann man nicht vermeiden, aber so unwahrscheinlich machen, dass sie **praktisch** keine Rolle spielen.

Benötigen einen Schaltkreis (**Synchronizer**), der die **Weiterleitung** der **fehlerhaften** Information eines metastabilen Flip-Flops **verhindert** und ihm Zeit gibt einen **stabilen Zustand** zu **erreichen**.

Idee: Verwende ein zusätzliches Flip-Flop SYN als Synchronizer:



182

Notizen

ANALYSE

Sei T_c die Periodendauer, T_{set} die Flip-Flop Setup-Time und T_{comb} die Verzögerung der kombinatorischen Logik.

Dann gilt für den Pfad **async** zum D-Eingang des Flip-Flops FF:

- Das Signal **async** wird um einen Takt verzögert.
- Das Signal **sync** muss $T_{set} + T_{comb}$ **vor** der nächsten (steigenden) Flanke stabil sein.
- Das Flip-Flop SYN hat also eine **Resolution Time** von $T_r = T_c - (T_{comb} + T_{set})$ zur Verfügung.

Sei $T_{set} = 2.5\text{ns}$, dann ergibt sich bei 50Mhz:

$$T_r = 20\text{ns} - (T_{comb} + 2.5\text{ns})$$

Das **optimale** T_r ergibt sich für $T_{comp} = 0$. Allerdings liegt T_{comp} durch das Design fest!

183

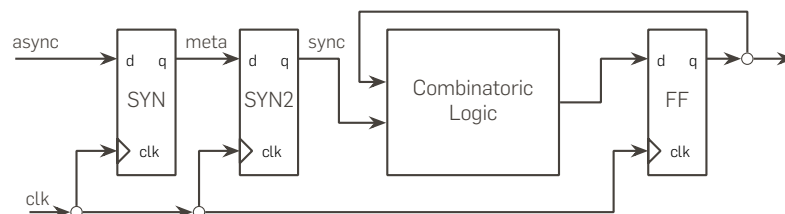
Notizen

ANALYSE (II)

Ist die kombinatorische Logik mit **1ns sehr schnell**, dann ergibt sich eine MTBF von **272 Jahren** (mit den schon verwendeten Parametern τ und ω).

Ist die kombinatorische Logik mit **12.5ns langsam**, dann beträgt die MTBF nur noch **0.8sec**.

Idee: Verwende zwei Flip-Flops für den Synchronizer:



Dann ergibt sich $T_r = 20\text{ns} - 2.5\text{ns} = 17.5\text{ns}$. Dies ergibt eine MTBF von **3000 Jahren**.

184

Notizen

BENUTZUNG VON SYNCHRONIZERN

Es ergeben sich folgende Regeln zur Benutzung von Synchronizern:

- Das asynchrone Signal sollte **frei von Glitches** sein.
- Ein Signal wird **an einem Ort** synchronisiert. Ein Synchronizer garantiert nicht, welcher Ausgabewert erreicht wird.
- **Vermeide** asynchrone **zusammenhängende Signale** (z.B. binäre Zahl). Alternative: Verwende Gray-Codes wenn möglich oder implementiere einen Message-Passing Mechanismus.
- **Bewerte** den Synchronizer nach jeder relevanten Designänderung **neu**.

Manche Technologien stellen Flip-Flops zur Verfügung, die gegen Metastabilität gehärtet sind.

185

Notizen

SYNCHRONIZER - EINE IMPLEMENTIERUNG

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Synchronizer is
5      port (clk      : in std_logic;
6            reset    : in std_logic;
7            async    : in std_logic;
8            sync     : out std_logic);
9  end Synchronizer;
10
11 architecture TwoFF of Synchronizer is
12     signal meta_reg, meta_next : std_logic;
13     signal sync_reg, sync_next : std_logic;
14 begin
15     process (clk, reset) -- Implementiere zwei Flip-Flops
16     begin
17         if (reset = '1') then
18             meta_reg <= '0';
19             sync_reg <= '0';
20         elsif (rising_edge(clk)) then

```

186

Notizen

SYNCHRONIZER - EINE IMPLEMENTIERUNG

```

1      meta_reg <= meta_next;
2      sync_reg <= sync_next;
3      end if;
4      end process;
5
6      -- Next state logic
7      meta_next <= async;
8      sync_next <= meta_reg;
9
10     -- Output logic
11     sync <= sync_reg;
12 end architecture;

```

Die Lösung mit zwei Flip-Flops ist **verbreitet**.

Es können auch spezielle gegen Metastabilität **gehärtete Flip-Flops** verwendet werden (die aber größer sind).

Evtl. kann man den **Takt künstlich verlangsamen** (vgl. PLL), wenn keine gehärteten Flip-Flops zur Verfügung stehen.

187

Notizen

CLOCK DOMAINS

In einem synchronen Schaltkreis werden **alle** Komponenten mit dem gleichen Takt betrieben. Alle Komponenten, die den gleichen Takt verwenden werden **Clock domain** genannt.

Aus verschiedenen Gründen kann man nicht immer mit **einer** Clock domain auf einem FPGA / ASIC arbeiten:

- Unterschiedliche physikalische I/O-Schnittstellen (z.B. Datenübertragung)
- Clock skew (Laufzeiten des Taktsignal führen zu einer Phasenverschiebung)
- Komplexität / Laufzeiten von Teilschaltkreisen (globaler Takt richtet sich nach dem langsamsten Schaltkreis)

Problem: An den Grenzen von Clock Domains müssen ganze Datenbusse auf die neue Domain synchronisiert werden. Dieses Problem ist als „**Clock Domain Crossing (CDC)**“ bekannt.

188

Notizen

CLOCK DOMAINS (II)

Idee: Benutzen für jedes Signal eines Busses einen eigenen Synchronizer.

FALSCH: Haben uns schon die Designregel „**Vermeide** asynchrone **zusammenhängende Signale** (z.B. binäre Zahl)“ in Zusammenhang mit Synchronizern überlegt.

Idee: Verwenden ein (synchronisiertes) Signal und führen ein **Handshaking-Protokoll** durch.

1. Der Sender gibt Daten aus und aktiviert **dann** das Signal REQ.
2. Der Empfänger übernimmt die Daten und aktiviert **danach** das Signal ACK.
3. Der Sender deaktiviert REQ und reaktiviert es nicht, solange bis auch ACK deaktiviert wurde.
4. Bemerkt der Empfänger, dass REQ deaktiviert wurde, so deaktiviert er auch ACK.

189

Notizen

CLOCK DOMAINS (III)

Dieses Vorgehen ist als 4-Phasen Handshake bekannt (es gibt auch effizienter Versionen wie den 2-Phasen Handshake).

Nachteil: Die Phasen brauchen Zeit, damit **geht (viel) Bandbreite** bei der Übertragung der Daten **verloren** (Handshake muss für jedes Datenpaket durchgeführt werden).

Idee: Verwenden einen asynchronen (dual-port) FIFO mit

- einem Write-Port mit eigenem Takt,
- einem Read-Port mit eigenem Takt und
- **full** bzw. **empty** Signal für die Flusskontrolle.

Vorteil: Durch dieses Vorgehen werden zeitaufwendige Handshakes vermieden (Datenrate nähert sich dem „Machbaren“ an).

190

Notizen

EIN ASYNCHRONER FIFO

Die folgenden Ideen stammen aus Peter Alfke, „Asynchronous FIFO in Virtex-II™ FPGAs“, TechXclusives, und der VHDL-Implementierung² von Alexander Pham.

Ziele:

- Es soll ein dual-ported RAM (BlockRAM) für einen FIFO-Speicher verwendet werden.
- Der read-Port und write-Port (bzw. das empty-Signal / full-Signal) können sich in **unterschiedlichen** Clock-Domains befinden.
- Es gibt je einen Read-Counter und Write-Counter für die Lese- bzw. Schreibadressen im FIFO

²http://www.asic-world.com/examples/vhdl/asyn_fifo.html

Notizen

EIN ASYNCHRONER FIFO (II)

Problem: Zur Generierung der Signale `empty` bzw. `full` werden die **read- und write-Adresse benötigt**. Es tritt also ein **Clock-Domain Crossing** auf. Aber: Wir können die Adressen nicht mit Synchronizern synchronisiert!

Idee: Verwende die **Gray-Codierung** für die Schreib- und Leseadressen.

Problem: Es sollen *alle* Speicherstellen des FIFOs benutzt werden können. Wenn read-Adresse gleich write-Adresse ist, dann kann der FIFO **leer** oder **voll** sein!

Idee: Teile den FIFO (Ringpuffer) in (vier) Sektoren ein.

- Befindet sich der **read**-Zeiger im Sektor **vor** dem **write**-Zeiger, dann wird eine **empty-Warnung** erzeugt.
- Wenn der **write**-Zeiger im Sektor **vor** dem **read**-Zeiger ist, dann generiere eine **full-Warnung**.

Notizen

EIN ASYNCHRONER FIFO (III)

Teilt man die Folge der Gray-Codes in vier gleich große Teile ein, so ergibt sich die folgende Liste der beiden hochwertigsten Bits:

00, 01, 11, 10, 00 (das letzte Paar wurde wiederholt)

Beobachtung: Zwei Paare (a_n, a_{n-1}) und (b_n, b_{n-1}) folgen aufeinander genau dann, wenn $a_{n-1} = b_n$ **und** $a_n \neq b_{n-1}$ gilt. Dies führt zu folgenden VHDL-Anweisungen:

```

1  fullWarn <= (writeAdr(writeAdr'high - 1) xnor
2                readAdr(readAdr'high)) and
3                (writeAdr(writeAdr'high) xor
4                readAdr(readAdr'high - 1));
5
6  emptyWarn <= (writeAdr(writeAdr'high - 1) xor
7                readAdr(readAdr'high)) and
8                (writeAdr(writeAdr'high) xnor
9                readAdr(readAdr'high - 1));

```

193

Notizen

EIN ASYNCHRONER FIFO (IV)

Der FIFO ist **leer** (bzw. **voll**), wenn eine Leerwarnung (bzw. Vollwarnung) gespeichert wurde und Leseadresse gleich Schreibadresse ist.

Frage: Wie speichert man eine Leerwarnung / Vollwarnung?

```

1  except : process(fullWarn, emptyWarn, clear)
2  begin
3      -- fifo should be cleared or we have an empty warning
4      if ((clear = '1') or (emptyWarn = '1')) then
5          -- '0' indicates a possible empty in the next future
6          exCase <= '0';
7      elsif (fullWarn = '1') then
8          -- '1' indicate a possible full in the next future
9          exCase <= '1';
10     end if;
11 end process;
12
13 -- Generate asynchronous empty / full signals
14 aFull  <= exCase and equalAdr;
15 aEmpty <= not exCase and equalAdr;

```

194

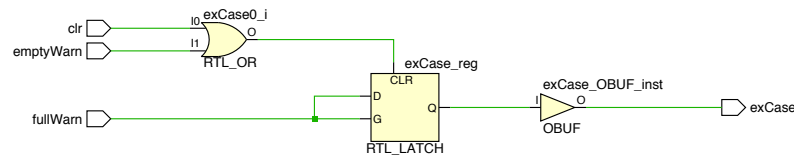
Notizen

EIN ASYNCHRONER FIFO (V)

An dieser Stelle verwenden wir (gewollt) ein **Latch** für **exCase**!

Dies erkennt man leicht, da in der **if**-Anweisung **nicht alle Fälle ausdekodiert** sind. Damit ergibt sich für **exCase** eine zeitliche Abhängigkeit, die mit der Instantiierung eines Latches (Speicher) gelöst werden muss, da das **Speicherverhalten nicht vom Takt abhängig** ist.

Dies zeigt auch die Synthese:



195

EIN ASYNCHRONER FIFO (VI)

Nur eine **Schreibaktion** bewirkt, das das full-Signal '1' wird. Damit ist die **steigende Flanke** synchron zur write-clock.

Analog: Die steigende Flanke des empty-Signals ist synchron zur read-clock.

Problem: Müssen noch die fallenden Flanken von „empty“ und „full“ synchronisieren:

```

1  synchronize_full : process (wClk, aFull)
2  begin
3      if(aFull = '1') then
4          -- happens synchronously (aFull = '1' needs equalAdr = '1')
5          full <= '1';
6      elsif (rising_edge(wClk)) then
7          -- Fifo is not full and the falling edge is synchronized too
8          full <= '0';
9      end if;
10 end process;

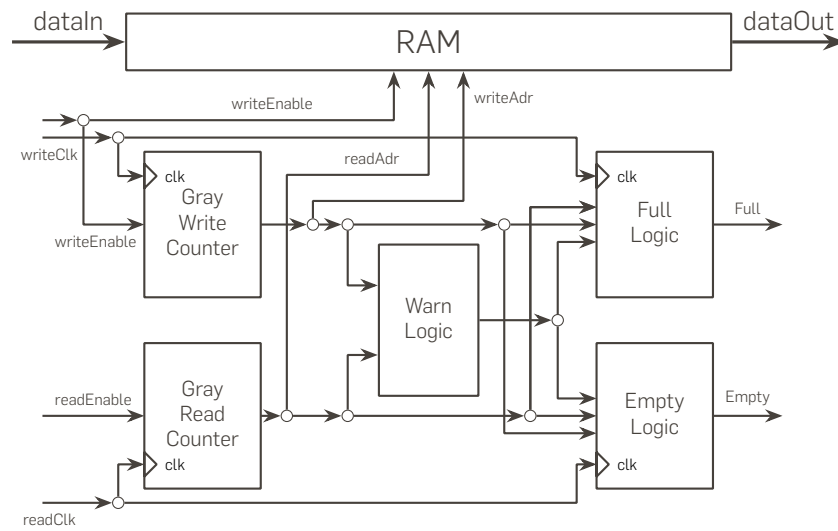
```

196

Notizen

Notizen

ASYNCHRONER FIFO (BLOCKSCHALTBIID)



197

Notizen

Notizen
