FH-OÖ/Hagenberg/HSD
Chip-Design 3 — CHD3
Markus Pfaff © 2025
Übung 8
Timing-Analyse, Asynchrone Antiquitäten

1. Aufgabe *Timing-Analyse*

Bislang wurde eine Timing-Analyse nur für kombinatorische Pfade zwischen Ein- und Ausgängen durchgeführt. In der Praxis des Digitalentwurfs sind solche Pfade jedoch seltene Ausnahmen. Die große Mehrzahl der Pfade beginnt an einem Flip-Flop-Ausgang und endet an einem Flip-Flop-Eingang. Die maximale Durchlaufzeit (meist etwas salopp als Länge des Pfades bezeichnet) durch die Kombinatorik zwischen zwei Flip-Flops ergibt sich aus dem Takt mit dem die beiden Flip-Flops getaktet werden.
Welche Zeitdauern müssen von der Dauer einer Taktperiode subtrahiert werden, um die maximal zulässige Durchlaufzeit zu berechnen?
Kann man diese Dauer auch dann berechnen, wenn die beiden Flip-Flops, die den Pfad einrahmen mit *unterschiedlichen* Takten betrieben werden? Dies kann nur unter bestimmten Bedingungen möglich sein, denen die beiden Taktsignale entsprechen müssen. Welche Bedingungen würden Sie stellen?
Der weitaus (!) einfachere Fall ist gegeben, wenn alle Flip-Flops in einem Design mit demselben Takt betrieben werden. Dann ist die maximale Länge aller Pfade leicht zu berechnen. Diese Berechnung wird von der Timing-Analyse (TimeQuest) automatisch durchgeführt. Hierbei wird auch der *Clock Skew* berücksichtigt, sofern Daten aus dem Layout der Schaltung zur Verfügung stehen.
Der Zähler aus der vorigen Übung soll nun mit Angaben zur Timing-Analyse ergänzt werden. Führen Sie in Quartus zunächst den Schritt *Compile* für dieses Design vollständig durch.
Im '*Report*, den Quartus erstellt, ist ein Bereich namens „TimeQuest Timing Analyzer“ zu finden. Einer der Unterpunkte namens *Clocks* stellt alle Takte dar, die Quartus im Design gefunden hat. In unserem Fall ist dies nur `iClk`. Welche Taktfrequenz nimmt die Timing-Analyse laut der dort zu findenden Tabelle für diesen Takt an? Diese Forderung kann das FPGA nicht erfüllen. Woran kann man dies ablesen? Wie nah kommt das FPGA der Forderung?
Nun soll eine eigene, realistischere Einschränkung (*Constraint*) für die Taktfrequenz angegeben werden. Legen Sie hierzu im Verzeichnis `quartus` eine Textdatei namens `Counter.sdc` an, die folgenden Inhalt hat:

```
# Define clocks used in design. Time spans are given in ns.
create_clock -name iClk -period 20 [get_ports {CLOCK_50}]

# Make TimeQuest calculate and apply clock skew
derive_clock_uncertainty
```

Diese Datei muss mittels des Menüpunktes *Project—Add/Remove Files in Project...* zum Quartus-Projekt hinzugefügt werden.
Ein erneuter Compile-Durchlauf in Quartus ergibt ein deutlich weniger rot eingefärbtes Ergebnis der Timing-Analyse. Ganz verschwunden sind die Probleme jedoch noch nicht („Unconstrained Paths“). Welche Pfade sind nach dem momentanen Stand noch nicht in der Timing-Analyse

berücksichtigt? Um Angaben für diese Pfade machen zu können, müssen genaue Informationen über das Zeitverhalten des Platinenlayouts und die am FPGA angeschlossenen Schaltungen vorhanden sein – eine nicht-triviale Aufgabe.

Asynchrone Antiquitäten In den 1970er und den frühen 80er Jahren war der Digitalentwurf hauptsächlich auf die Realisierung durch TTL-ICs beschränkt. Damit waren gemessen an der Funktionalität hohe Kosten verbunden. Es wurde damals auch auf asynchrone Schaltungsdesigns zurückgegriffen, um an der ein oder anderen Stelle einige Gatter einzusparen. Oft wurde weder simuliert noch wurde das *Timing* der Schaltungen detailliert untersucht. Stattdessen hat man Schaltungen häufig durch Praxistests erprobt, was sich aber immer nur auf spezielle Betriebssituationen beziehen kann und daher keineswegs eine ausreichende Verifikationsmethode darstellt. Leider haben sich einige dieser zweifelhaften „Tricks“ (insbesondere bei langjährigen PraktikerInnen) bis heute erhalten. Die nachfolgenden Aufgaben sollen einige verbreitete Fallen aufzeigen.

2. Aufgabe *Don't! – Missbrauch des asynchronen Reset-Eingangs*

Der asynchrone Eingang *Clear* oder *Reset* von Flip-Flops sollte nur für die Initialsierung von Flip-Flops verwendet werden. Selbst dann ist noch Vorsicht geboten, weil das betreffende Signal `inResetAsync` einsynchronisiert werden muss. So kann garantiert werden, dass beim Deaktivieren des Signals keine Zweikomponentenübergänge am Flip-Flop auftreten. Schließlich könnte sich das Signal `Clk` zeitnah mit `inResetAsync` ändern. Diese Problematik wird im folgenden Semester genauer untersucht.
Eine Verwendung des asynchronen Reset-Eingangs im normalen Betrieb einer Schaltung, also etwa um das Register eines dekadischen Zählers (0-9) auf den Wert 9 folgend auf Null zu setzen, ist sehr problematisch und sollte vermieden werden. Hier ein Beispiel:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.global.all;
5
6  entity CounterAsyncZero is
7    generic(
8      gClkFrequency : natural := 50E6);
9    port (
10     inResetAsync : in  std_ulogic;
11     iClk         : in  std_ulogic;
12     oCountedTo   : out unsigned(LogDualis(gClkFrequency)-1 downto 0));
13 end entity CounterAsyncZero;
```

```
1  architecture Rtl of CounterAsyncZero is
2    signal Counter : unsigned(oCountedTo'range) := (others => '0');
3  begin
4
5    process (iClk, inResetAsync, Counter) is
6    begin
7      if inResetAsync = not('1') or to_integer(Counter)=gClkFrequency then
8        Counter <= (others => '0');
9      elsif rising_edge(iClk) then
10       Counter <= Counter+1;
```

```
    end if;
  end process;

  oCountedTo <= Counter;

end architecture Rtl;
```

Mit welcher Rate (Anzahl pro Sekunde) tritt bei diesem Zähler der Wert 0 auf?
Synthetisieren Sie diesen Entwurf in Quartus. Sind ernstzunehmende *Warnings* zu verzeichnen? Probleme treten bei diesem Entwurf in folgenden Bereichen auf:

- Beim Wechsel vom maximalen Zählerwert zum Wert 0 tritt kurz ein Zwischenwert auf. Zeigen Sie dies in der Simulation. In der Realität wird dieser Zwischenwert während einer gewissen Zeitdauer anliegen. Wie lange ist diese Dauer dagegen in der Simulation?
- Das Signal, welches an den asynchronen *Clear*-Eingängen der Flip-Flops angeschlossen ist, darf keine statischen *Hazards* aufweisen. Warum nicht? Garantieren VHDL-Entwurf bzw. Synthese diese Bedingung?
- Kann die Timinganalyse die entstehende Schaltung überhaupt beurteilen?

Erstellen Sie einen *synchronen* Entwurf (VHDL), der die eigentlich gewünschte Funktion des obigen VHDL-Codes korrekt umsetzt. Optional können Sie die Funktion ihres Entwurfs auf dem Board zeigen.

3. Aufgabe *Don't! – Missbrauch von Takteingängen*

Im folgenden Entwurf wurde versucht, einen Tastendruck mittels `'event` auszuwerten:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CounterKeyEventZero is
  generic(
    -- Clock: 50 MHz : 26 bits are needed to make the MSB cycle through its
    -- pace in more than 1 s
    gCounterBitLength : natural := 26);
  port (
    inResetAsync : in  std_ulogic;
    iClk         : in  std_ulogic;
    iKey         : in  std_ulogic;
    oCountedTo   : out unsigned(gCounterBitLength-1 downto 0));
end entity CounterKeyEventZero;
```

```
architecture Rtl of CounterKeyEventZero is

  signal Counter : unsigned(oCountedTo'range) := (others => '0');

begin

  process (iClk, iKey, inResetAsync) is
  begin
    if inResetAsync = not('1') then
```

```
10        Counter <= (others => '0');
11      elsif rising_edge(iClk) then
12        Counter <= Counter+1 mod 2**gCounterBitLength;
13      elsif rising_edge(iKey) then
14        Counter <= (others => '0');
15      end if;
16    end process;
17
18    oCountedTo <= Counter;
19
20  end architecture Rtl;
```

Was auf den ersten Blick sinnvoll erscheint, entpuppt sich als für das Synthesewerkzeug äußerst schwer verdaulich. Welche Fehlermeldung erhalten Sie? Warum kann das Synthesewerkzeug diesen VHDL-Code nicht umsetzen?
Erstellen Sie einen synchronen Entwurf, der die eigentlich gewünschte Funktion des obigen VHDL-Codes korrekt umsetzt, also bei Tastendruck den Zählerwert auf Null setzt. Optional können Sie die Funktion ihres Entwurfs auf dem Board zeigen.

4. Aufgabe *Don't! Gated Clock*

Häufig soll eine Schaltung ihren Zustand nur selektiv bei bestimmten Taktflanken wechseln, nicht jedoch bei jeder Taktflanke. Im folgenden Beispiel soll ein Zähler durch einen Eingang entweder zum Zählen freigegeben werden oder sein Zählerwert soll „eingefroren" werden:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity CounterGatedClk is
6    generic(
7      -- Clock: 50 MHz : 26 bits are needed to make the MSB cycle through its
8      -- pace in more than 1 s
9      gCounterBitLength : natural := 26);
10   port (
11     inResetAsync : in  std_ulogic;
12     iClk         : in  std_ulogic;
13     iEnable      : in  std_ulogic;
14     oCountedTo   : out unsigned(gCounterBitLength-1 downto 0));
15 end entity CounterGatedClk;
```

```
1  architecture Rtl of CounterGatedClk is
2
3    signal Clk     : std_ulogic;
4    signal Counter : unsigned(oCountedTo'range) := (others => '0');
5
6  begin
7
8    Clk <= iClk when iEnable = '1' else
9           '0';
10
11   process (Clk, inResetAsync) is
12   begin
13     if inResetAsync = not('1') then
```

```
14        Counter <= (others => '0');
15      elsif rising_edge(Clk) then
16        Counter <= Counter+1 mod 2**gCounterBitLength;
17      end if;
18    end process;
19
20    oCountedTo <= Counter;
21
22  end architecture Rtl;
```

Synthetisieren Sie diesen Entwurf in Quartus. Gibt es *Warnings*, die auf den problematischen Charakter des Entwurfs hinweisen?
Wie wird die Steuerung des Taktes durch die Synthese realisiert?
Wenn der Takt über eine eigene Kombinatorik an die Takteingänge der Flip-Flops in einer Schaltung herangeführt wird, spricht man von einem *gated Clock*. Mit solchen *gated Clocks* sind leider schwerwiegende Probleme verbunden:

- Je nachdem wann das Enable-Signal sich verändert, kommt es zu kurzen, sogar extrem kurzen Taktabschnitten. Daneben können Taktflanken mit einem – deutlich – zu kurzen Abstand auftreten.
- Wenn das Enable-Signal nicht frei von *Hazards* ist, kommt es zu weiteren Taktflanken, welche nicht im vorgesehenen zeitlichen Abstand liegen.
- Die Struktur des *Clock Tree* wird durch den Einbau zusätzlicher Kombinatorik gestört.
- Die zusätzliche Kombinatorik verzögert den *gated Clock* gegenüber dem eigentlichen Takt, was als *Clock Skew* zu einer Verschlechterung des Zeitverhaltens führt.

Um die gewünschte Funktion zu verwirklichen, verwendet man daher eine synchrone Schaltung. Bei einem Zähler wird beispielsweise ausgewählt, ob an den Eingang des Zustandsregisters (Register für den Zählerstand) einfach der momentane Zählerstand oder der um eins erhöhte Zählerstand angelegt wird.
Bauen Sie die obige Beschreibung entsprechend um und prüfen Sie das Syntheseergebnis.

Gated Clocks haben jedoch durchaus eine Anwendung. Der Vorteil von *gated Clocks* liegt darin, dass ein Flip-Flop, das keine Taktflanke bekommt, deutlich weniger Energie verbraucht als eines, dass eine Taktflanke bekommt. Dies gilt auch dann, wenn der Wert, der eingelesen wird derjenige ist, der ohnehin schon gespeichert war.
Auf Grund der Probleme, die beim *Clock-Gating* entstehen, ist es üblich, nicht etwa einzelne Flip-Flops, sondern ganze Schaltungsteile auf einen Schlag mit einem *gated Clock* zu versorgen. Dieser wird auch nicht einfach über ein UND-Gatter gesteuert, sondern mittels einer speziellen Zelle, durch welche zu kurze Taktzyklen vermieden werden. In FPGAs spielt der Energieverbrauch (derzeit) keine entscheidende Rolle, so dass *Clock gating* unüblich ist. Die dazu notwendige Zelle ist jedoch an der Wurzel jedes *Clock Trees* in Altera-FPGAs vorhanden und trägt den Namen *Clock Control Block*. Verwendet wird diese Zelle beispielsweise beim *Prototyping* von ASICs mittels FPGAs.