FH-OÖ/Hagenberg/HSD
Chip-Design 3 — CHD3
Markus Pfaff © 2025
Übung 4
Hierarchie, DE1-SoC, Multiplizierer, Tcl

1. Aufgabe *Hierarchie im Entwurf. Achtung! Nicht mit Quartus-Lite-Edition bearbeitbar!*

Wenn nichts anderes eingestellt ist, versucht *Quartus* das Design so gut wie möglich zu optimieren. Hierzu ist es notwendig, die (*Entity*-)Grenzen zwischen den einzelnen *units* zu entfernen, damit über diese Grenzen hinweg optimiert werden kann. Dies wird als Ausflachung oder *Flattening* bezeichnet und von Quartus automatisch vorgenommen. In der Lite Edition haben Sie gar keine andere Möglichkeit. Die Subscription-Edition bietet dagegen die Möglichkeit diejenigen *entity*-Grenzen auszuwählen, die Sie erhalten möchten.
Wenden wir uns nochmals der Ausgaben des *RTL-* und *Technology-Map Viewer* aus der letzten Übung zu. Öffnen Sie dazu das Projekt, welches das *Testbed* für die Multiplexer enthält. Im Schaltplan sieht man, dass die Optimierung von Quartus ganze Arbeit geleistet und alle Multiplexer in einer einzigen 3LUT untergebracht hat.

In der letzten Übung ist ein Entwurf erstellt worden der eine Hierarchie mehrerer *units* enthält, nämlich jener, der NAND-Gatter instantiiert. Allerdings sind die unterschiedlichen Multiplexer-Beschreibungen auch eine Unterebene des *Testbeds*. Diese Hierarchieebenen sollen nun erhalten bleiben. Im *Utility Window Project Navigator* wählen Sie den *Tab Hierarchy* aus. Dort öffnen Sie alle Hierarchieebenen des besagten Entwurfs. Nun wählen Sie alle *units* unterhalb der Ebene des *Testbeds* aus (also alle Realisierungen des Multiplexers und deren Unterebnen). Mit der rechten Maustaste im Auswahlbereich erscheint die Möglichkeit unter *Design Partition—Set as Design Partition* die Hierarchiegrenzen sozusagen einzufrieren. Beachten Sie das Symbol, das rechts neben den *units* erscheint. Bislang war dieses Symbol nur auf der obersten Ebene (*Top Level*) zu sehen.

Mit *Compile* lassen Sie Quartus das Design neu verarbeiten.

- Wie sehen nun die Ausgaben von *RTL-* und *Technology-Map Viewer* aus?
- Wie hat sich der Ressourcenverbrauch verändert?
- Wie haben sich die Durchlaufzeiten verändert?

Experimentieren Sie mit einer weiteren, frei gewählten Einstellung zur Erhaltung unterschiedlicher *entity*-Grenzen (einige einfrieren, andere nicht). Wie wirken sich diese Einstellungen auf das Optimierungsergebnis aus?
Warum könnte Altera diese Möglichkeiten ausgerechnet in die Subscription-Edition eingebaut haben, obwohl die Ergebnisse durchweg schlechter (langsamer und größer) werden? Des Rätsels Lösung findet sich bei Altera unter dem Stichwort *Incremental Compile*. Worum handelt es sich hierbei? Nah damit verwandt ist auch das Konzept *LogicLock*.

2. Aufgabe *Functionality Sharing bei einem Multiplexer*

Um die Auswirkungen der Priorisierung genauer zu betrachten, beschreiben Sie einen Multiplexer namens `Mpx`, der einen von n Eingängen, beispielsweise für $n = 12$ also `iA(0)` bis `iA(11)` auf einen Ausgang `oY` (*type* `std_ulogic`) schalten kann.
Welcher der n Eingänge dies ist, wird mittels des Auswahleingangs `iSel` festgelegt. Der Eingang `iSel` soll vom *type* `std_ulogic_vector` in der benötigten Länge ($\log_2 n$) sein. Denken Sie

sich diesen Vektor als vorzeichenlose Binärzahl, welche den Ausgang mit der passenden Nummer auswählt.
Der Multiplexer soll in 2 verschiedenen *architectures* einerseits mit *case statement* (Name der *architecture*: `UsingCase`) und andererseits mit *if-then-elsif-else statement* (Name der *architecture*: `UsingIf`) realisiert werden. Instantiieren Sie beide Beschreibung nebeneinander in einer eigenen *entity/architecture* `LargeMuxes(Rtl)`. Beide Beschreibungen teilen sich die gleichen Eingänge, verfügen jedoch über je einen eigenen Ausgang.
Untersuchen Sie das Syntheseergebnis für $n = 8$ im Hinblick auf die verbrauchten Ressourcen sowie im *RTL-* und *Technology-Map Viewer*. Werden Ihre Erwartungen hinsichtlich der Unterschiede zwischen den beiden Entwürfen erfüllt?

Wiederholen Sie diese Untersuchung nun für $n = 12$. Quartus erzeugt ein bemerkenswert kompaktes Ergebnis. Untersuchen Sie dieses insbesondere mit dem *Technology-Map Viewer*. Vergleichen Sie die Verzögerungszeiten für die beiden Varianten.

Es liegt nahe, n mittels eines *generics* im VHDL-Modell darzustellen. Sie brauchen aber kein allgemeingültiges Modell zu erstellen, das beliebige Werte von n zulässt. Es reicht aus, lediglich die Werte $n = 8$ und $n = 12$ zu berücksichtigen. Wodurch würde es erschwert werden, ein solches allgemeingültiges Modell zu erstellen? Ist dies auf Basis von *if-* oder *case-*Anweisungen überhaupt möglich?
Sie können beispielsweise nur diese Werte für den *generic* zulassen, indem Sie dies per *assertion statement* prüfen. Einen *subtype* von `integer`, der nur die beiden Zahlen 8 und 12 umfasst, können Sie leider in VHDL nicht definieren.
Die Implementationen für den Wert 8 und den Wert 12 können Sie mittels zweier *conditional generate statements* auswählen.

3. Aufgabe *Schaltplan DE1-SoC*

- *Sheet* 8 des Schaltplans (*Decoupling*) ist recht eintönig. Welchen Sinn haben die vielen dort dargestellten Kondensatoren? Können Sie einige der Kondensatoren auf dem Board finden?
- *Sheet* 12 und 13 stellen die Beschaltung der GPIO-Buchsen (*General Purpose Input/Output*) dar. Die Bauelemente, die dort zu sehen sind, dienen dem Schutz des FGPAs vor zu hohem Strom, zu hoher positiver oder zu hoher negativer Spannung. Gemeint sind die Bauelemente RNx (47 Ω) und Dx (Doppel-Schottky-Diode BAT54S). Warum bewirken diese Bauelemente in der gegebenen Schaltung einen solchen Schutz?
- Auf *Sheet* 21 findet sich die Schaltung für die Infrarot-LED (`IE_Emitter_LED`). Seltsamerweise werden zwei Widerstände von 4,99 Ω parallelgeschaltet, die doch auch durch einen Widerstand von 2,5 Ω ersetzt werden könnten. Der Grund liegt im Leistungsumsatz innerhalb jeden der beiden Widerstände. Berechnen Sie diesen. Gehen Sie von $U_{CE} = 0.2\text{V}$ für den gesättigten Transistor `Q2` aus.
- Suchen Sie die beiden Widerstände auf dem Board (rechts unten). Vergleichen Sie deren Baugröße mit derjenigen der anderen Widerstände. Es handelt sich um die Baugröße 1206 (Länge: $12\frac{1}{1000}$inch, Breite:$6\frac{1}{1000}$inch). Widerstände dieser Baugröße können bei Raumtemperatur ohne Zwangslüftung maximal 0,25 W an elektrischer Leistung als Wärme abführen. Darf die LED dauerhaft eingeschaltet sein?
- Welche Leistung wird in der Infrarot-LED umgesetzt? Deren Baugröße ist erstaunlich klein. Warum überhitzt die LED nicht?
- Wozu dienen die Bauelemente ab *Sheet* 27?

4. Aufgabe *Fakultativ: Multiplizierer nach dem Add-Shift-Verfahren*

Ein Multiplizierer kann in VHDL durch mehrere Verfahren beschrieben werden. Das einfachste ist die Verwendung des Operators `"*"`. In dieser Aufgabe soll das Add-Shift Verfahren zum Einsatz kommen. Der Produktoperator `"*"` darf nicht verwendet werden (nur später für den Vergleich).

Die binäre Multiplikation vorzeichenloser Zahlen an einem Beispiel:

```
Multiplicand Md  (14)      1110
Multiplier   Mr  (11)  x   1011
                           --------
                               1110
                              1110
                             0000
                            1110
                           --------
Product P       (154)  10011010
```

Es gibt aber eine effizientere Alternative:

```
       1011 x 1110      dezimal 11 x 14
PP     ||||   0000
  1110*|||1=  1110
       |||    ----      +
       |||   01110
       |||   -----      >> (shift)
PP     |||    01110
  1110*||1=   1110|
       |||    ----|     +
       ||    101010
       ||    ------     >> (shift)
PP     ||     101010
  1110*|0=    0000||
       |      ----||    +
       |     0101010
       |     -------    >> (shift)
PP     |      0101010
  1110*1=     1110|||
              ----|||   +
             10011010
             --------   >> shift
PP            10011010  PP=Ergebnis, dezimal 154
```

Anmerkung: PP ist das *Partial Product* mit dem Startwert 0
Der Vorteil dieser Methode liegt darin, dass ein Addierer geringerer Bitbreite verwendet werden kann. Bei der Schiebeoperation werden nun zwar mehr Stellen bewegt, aber diese ist in Hardware nur mit geringem Aufwand verbunden. In Summe führt dies zu einer Implementierung mit geringerem Flächenbedarf. Sie können beliebig eine der beiden Methoden für Ihre Implementation(en) auswählen.

Legen Sie zunächst eine neue *unit* `unitMultiplier` (inkl. aller notwendigen Unterverzeichnisse) für den Multiplizierer im Verzeichnis `grpArithmetics` an.

Erstellen Sie die *entity* für einen parametrisierbaren Multiplizierer (großteils schon vorgegeben). Für die Implementierung gilt also: Beide Faktoren `iMultiplicand` und `iMultiplier` sind vom Typ *type* `unsigned(gArgRange-1 downto 0)` aus dem *package* `numeric_std`. Das Produkt `oProduct` ist ebenfalls vom Typ *type* `unsigned`, wodurch sich der Wertebreich automatisch ergibt. Der *generic* `gArgRange` soll zu Beginn den Wert 4 erhalten. Verwenden Sie einen passenden *type* für `gArgRange`.

Die VHDL-Implementierung selbst kann unter Verwendung verschiedener Konstrukte erfolgen. Im Rahmen dieser Übung beschränken wir uns auf zwei konkrete Implementierungen.

Für die erste Implementierung erstellen Sie eine *architecture* `UsingForLoop` eines Multiplizierers nach dem zuvor vorgestellten verbesserten Add-Shift Verfahren. Verwenden Sie für das Add-Shift Verfahren das *for-loop statement*.

Für die zweite Implementierung erstellen Sie eine *architecture* `UsingForGenerate` eines Multiplizierers wiederum nach dem gleichen Add-Shift Verfahren. Verwenden Sie für das Add-Shift Verfahren das *for-generate statement*.

Instanziieren Sie ihre beiden Beschreibungen sowie die bereits vorgegebene *architecture* `UsingOperator` aus einer *Testbench* heraus. Prüfen Sie die Funktion Ihrer Beschreibungen mittels Simulation. Automatisieren Sie die Verifikation! Die Testbench sollte also bei erfolgreichem Durchlauf eine entsprechende Meldung ausgeben und bei einem Fehler die Simualation automatisch stoppen. Zur Prüfung der Richtigkeit der Ergebnisse können Sie z.B. die Resultate der verschiedenen Implementierungen untereinander vergleichen, wobei bei der Verwendung des Multiplikationsopterators von der Richtigkeit der Berechnung ausgegangen werden kann. Überlegen Sie auch, ob und wie Sie die korrekte Funktion über den gesamten Wertebereich beider Eingabewerte prüfen können.

Untersuchen Sie das Syntheseergebnis von *architecture* `UsingForLoop`, `UsingForGenerate` und *architecture* `UsingOperator` hinsichtlich RTL- und Technology-Schematic, Flächenbedarf und Durchlaufzeit. Überprüfen Sie insbesondere Ihre Erwartungen hinsichtlich der Unterschiede zwischen den Entwürfen.

Stellen Sie den *generic* `gArgRange` auf 16 ein und prüfen Sie die Funktion Ihrer Beschreibung für den parametrisierbaren Multiplizierer mittels Simulation.

Synthetisieren Sie die erstellten Beschreibungen `UsingForLoop`, `UsingForGenerate` bzw. `UsingOperator`. Welche Endergebnisse (Area Report, Timing-Analyse) erhalten Sie für `gArgRange = 16` welche für `gArgRange = 32`?

Hinweis: Sie können die *generics* der *top level entity* im Quartus-Projekt einstellen und müssen dazu nicht etwa den VHDL-Code ändern.

QSF Eintrag „`set_parameter -name gArgRange 24`“ in die qsf-Datei eintragen.

GUI Wählen Sie im *Quartus*-Menü *Assignments—Setting...—Compiler Setting—Default Parameters*. Geben Sie im Feld *Name:* `gArgRange` ein und im Feld *Default settings:* den gewünschten Wert ein. Klicken Sie auf *Add* und bestätigen Sie anschließend mit *OK*.

Bei einem neuerlichen Synthesedurchlauf wird nun der auf diese Weise eingestellte Wert verwendet. Wie wirken sich die verschiedenen Einstellung auf Geschwindigkeit und Flächenbedarf aus?

Selbstverständlich können Sie eine (oder auch beide) Ihrer Beschreibungen nun auch auf dem Board ausprobieren, indem Sie sie im Spezialrechenwerk aus der vorigen Übung verwenden.

5. Aufgabe *Fakultativ: Scripting Quartus*

Quartus kann sowohl mittels der graphischen Oberfläche als auch durch Befehle in der Sprache TCL gesteuert werden. Diese Befehle lassen sich einerseits interaktiv an Quartus geben (beispielsweise in der TCL-*Console* in Quartus) andererseits kann man sie aber auch in einer Datei zusammenfassen. Eine solche Datei wird auch Script genannt. Ein einfaches Beispiel für ein solches Script finden Sie in `Uebung04/prjUebung04/grpStatements/unitCaseExample/quartus/Compile.tcl`. Nachfolgend ist diese Datei wiedergegeben:

```
# Set a TCL variable containing the top level entity name
set DesignName CaseExample

# Quartus spreads TCL statements over packages which we have to load
# before the statements can be used
load_package project
load_package flow

# Create a new Quartus project (and delete any existing at the same time)
project_new -overwrite $DesignName

# Which FPGA will be used?
set_global_assignment -name FAMILY "Cyclone␣V"
set_global_assignment -name DEVICE 5CSEMA5F31C6

# Set the top level entity
set_global_assignment -name TOP_LEVEL_ENTITY $DesignName

# Name all (!) files which are part of the project. A relative path
# name is used starting from the directory quartus_sh is started in.
set_global_assignment -name VHDL_FILE ../src/$DesignName-ea.vhd

# Set the name of the directory there results shall be stored.
set_global_assignment -name PROJECT_OUTPUT_DIRECTORY output_files

# A command residing in the flow package is used, namely analysis &
# compilation
execute_flow -analysis_and_elaboration

# Alternatively a complete compile could be done.
#execute_flow -compile

# Before the script ends close the Quartus project.
project_close
```

Die Kommentare sollen eine erste, nicht in die Tiefe gehende Erläuterung geben. Genauere Informationen geben das *Quartus II Handbook* (Abschnitt *TCL Scripting* und das *Quartus II Scripting Reference Manual*, die beide als kostenloser Download bei Altera erhältlich sind.
Die Fähigkeiten von TCL werden in diesem Script nur genutzt, um die Variable DesignName zu definieren und deren Wert mit Hilfe von $DesignName abzufragen. Ansonsten werden lediglich spezifische Quartus-Befehle aufgerufen. TCL bietet jedoch einen deutlich größeren Sprachumfang und daneben mit der Erweiterung TK (beide werden gemeinsam als Tcl/Tk bezeichnet) auch die Möglichkeit einfache graphische *User Interfaces* zu erzeugen. Tcl/Tk ist im EDA-Bereich weit verbreitet, so dass es sich lohnt, zumindest die Grundzüge der Sprache zu erlernen. Komplexere Aufgaben sollte man jedoch nicht mit Tcl/Tk lösen, dazu eignet sich beispielsweise Python deutlich besser.
Das Script `Compile.tcl` kann auf unterschiedliche Arten gestartet werden. Beispielsweise von ei-

nem *Windows Command Window* mit der Befehlszeile:
`%QUARTUS_ROOTDIR%\\bin64\quartus_sh.exe --script Compile.tcl`, wobei
`%QUARTUS_ROOTDIR%` eine System-Umgebungsvariable (Windows *Environment Variable*) ist, die auf den Pfad der Quartus-Installation gesetzt ist.
Im Verzeichnis `Uebung04/prjUebung04/grpStatements/unitCaseExample/quartus/` findet sich auch ein *Windows Shortcut*, der diese Befehlszeile startet. Allerdings wird das *Windows Command Window* automatisch bei Scriptende und genauso bei Fehlern geschlossen. Starten Sie die Quartus-*Shell* nun mittels Doppelklick auf den *Shortcut*.
Nach Aufruf des Scripts wird von Quartus ein Projekt generiert (*File Extension* .qpf), dass auch in der graphischen Oberfläche geöffnet werden kann. Öffnen Sie das Projekt in der graphischen Oberfläche um dies nachzuprüfen. Häufig werden Projekte mittels Scripting generiert um anschließend die graphische Oberfläche für die Analyse und schrittweise Verbesserung zu verwenden.