FH-OÖ/Hagenberg/HSD
Chip-Design 3 — CHD3
Markus Pfaff © 2025
Übung 10
Generierter Takt, Gesamt-Design, VUnit

OBERÖSTERREICH

Modelsim-Tipps:

- Modelsim lässt sich mit Eingabe von `exit -f` in der TCL-Shell ohne den lästigen Nachfragedialog verlassen.
- Die Taste 'f' im Waveform-Window löst einen *Full-Zoom* aus.

1. Aufgabe *Problematisch! – Generierte Taktsignale*

Statt der Verwendung eines *Strobe*-Signals liegt die Erzeugung eines neuen Taktsignals mit niedrigerer Frequenz nahe. Dies ist jedoch – bestenfalls – mit Einschränkungen empfehlenswert:

- Es existieren dann im System zumindest zwei Takte (die aber wenigstens noch in einer festen Zeitbeziehung stehen). Die Flanken des generierten Taktes werden stets zeitlich hinter den Flanken des ursprünglichen Taktes liegen. Warum?

  Die Zeitbeziehung zwischen Ursprungstakt und generiertem Takt wird bereits durch die Verzögerungszeiten bei der Taktgenerierung ungenau. Hinzu kommt, dass man nun einen zweiten *Clock Tree* für den (bzw. jeden) neuen Takt benötigt, was bei wenigen Takten noch möglich sein mag, jedoch nicht mehr bei wirklich vielen Takten. Die unterschiedlichen *Clock Trees* werden untereinander selbstverständlich wiederum Zeitverschiebungen aufweisen.

- Stellt das Subsystem, welches mit dem langsameren Takt arbeitet, dem schneller getakteten System Ergebnisse seiner Berechnungen zur Verfügung, muss deren Verfügbarkeit entweder durch Mitzählen im schnelleren System oder durch entsprechende *Handshake*-Signale sichergestellt werden.

- Beim umgekehrten Weg müssen die Signale aus dem schneller getakteten System lange genug konstant gehalten werden, damit das langsamere System sie auch abspeichern kann.

  Besonders problematisch ist jedoch der Umstand zu sehen, dass die Taktflanken des sendenden Systems nun knapp vor denen des empfangenden Systems liegen. Wie kann es hierdurch zu Problemen kommen? Verwenden Sie ein *Timing*-Diagramm zur Verdeutlichung!

Aus den genannten Gründen sollte man durch Zähler oder ähnliche FSM-generierte Takte in digitalen Schaltungen möglichst vermeiden und wenn überhaupt, dann nur für wenige FF, die nah beieinander liegen verwenden.
Deutlich einfacher wird der Umgang mit mehreren Takten, wenn diese aus einer Quelle kommen, welche mehrere Takte mit fixierter Phasenlage ausgeben kann, beispielsweise aus einer

PLL (hierzu in den kommenden Semestern mehr). Allerdings ist auch in diesem Fall die Problematik der hohen maximalen Verzögerungszeit der *Clock-Trees* und deren relativ große Streuung zwischen minimaler und maximaler Verzögerungszeit noch immer ein Problem.
In der Praxis ist daher die Verwendung von *Strobe*-Signalen die einfachste und zugleich effizienteste Variante.

2. Aufgabe *Reaktionszeit-Spiel für zwei Personen*

Bei unserem Spiel soll Spieler(in) A so schnell wie möglich ihre/seine Taste (heißt ebenfalls A) drücken, um darauf zu reagieren, dass Spieler(in) B ihre/seine Taste B gedrückt hat.

Die Ausgabe der Reaktionszeit in Tausendstel Sekunden erfolgt auf drei Siebensegmentanzeigen. Der Spielstatus wird über LEDs angezeigt, wobei jedem Zustand eine LED zugeordnet ist.

- Zu Beginn des Spieles zeigt die Siebensegmentanzeige 0 an. Um das Spielgerät in Bereitschaft zu versetzen, muss Taste A gedrückt werden.
- Druck auf Taste B startet die Zeitnehmung (Tausendstel-Sekunden-Zähler). Die Siebensegmentanzeige zeigt fortlaufend alle Stellen der seit dem Tastendruck vergangenen Zeitdauer an.
- Spieler/in A antwortet so schnell wie möglich durch Drücken von Taste A. Der Zähler stoppt daraufhin. Die Anzeige gibt wie immer auch jetzt den aktuellen Zählerstand wieder.
- Spieler/in B kann mit Drücken von Taste B den Zähler (und damit auch die Anzeige) auf 0 zurücksetzen. Das Spiel befindet sich im gleichen Status wie am Anfang.

Implementierungsvorschlag

- Ein Strobegenerator, der alle $1/1000$ s einen Impuls erzeugt, dient als Zeitbasis.
- Drei 4-Bit-Modulo-10-Zähler (zählen also jeweils von 0 bis $(10-1)=9$) werden verwendet, um die Reaktionszeit in $1/1000$ s-Schritten zu zählen. Der Zähler für die niederwertigste Stelle inkrementiert (modulo 10) nur bei jener steigenden Taktflanke bei welcher der Ausgang des Strobegenerators *und* das `iEnable`-Signal von der Steuerungs-FSM gesetzt sind.

  Die Modulo-10-Zähler für die höherwertigen Stellen inkrementieren nur, wenn die vorherige Stelle überläuft. Alle Zähler verfügen über einen synchronen Eingang `iZero`, der von der FSM gesteuert wird und dem synchronen Setzen des Zählerstandes auf 0 dient.
- Die Siebensegmentanzeigen zeigen den aktuellen Zählerstand des zugehörigen Zählers an. Sie sind also einfach über Binär-zu-7Segment-Decoder mit den 4-Bit-Modulo-10-Zählern verbunden.
- Die Status-LED(s) werden unmittelbar durch den ihnen entsprechenden Zustand zum Leuchten gebracht. Jeder Zustand ist einer LED zugeordnet.
- Im Kern der Implementation der Spielfunktion befindet sich eine Finite-State-Machine (FSM), welche die beiden Eingänge von den Tasten (`iA`, `iB`) auswertet, und die Zähler mit entsprechenden Befehlssignalen ansteuert. Vorschlag für diese FSM:

  - Inputs: `iA`, `iB`
  - Outputs: `oZeroCounter`, `oEnableCounter`, `oLed` (Array aller LEDs)
  - States:

    Locked: Zählerstand auf 0 setzen. Warten auf Taste A.

    Unlocked: Spieler/in A ist bereit. Warten auf Taste B.

    CountUpTime: Zähler laufen. Warten auf Taste A.

    ShowResult: Zähler stoppen. Warten auf Taste B.

- Die Tasten stellen asynchrone Eingangssignale dar, welche auf den Systemtakt einsynchronisiert werden müssen.

Zunächst erstellen Sie bitte eine FSM zur Spielsteuerung und dokumentieren Sie diese in Form eines Bubble-Diagramms. Handelt es sich um eine Moore- oder eine Mealy-FSM?

Erstellen Sie ein VHDL-Modell `ReactionTimeGameFsm` dieser FSM. Vergessen Sie nicht, zur Kommunikation mit dem Zeitzähler die entsprechenden Signale vorzusehen. Handelt es sich beim VHDL-Modell tatsächlich um die vorher festgestellte Art der FSM (Mealy oder Moore)?

Weisen Sie die Funktion mittels Simulation nach.
Wie hoch ist der Ressourcenverbrauch in der Synthese? Mit welcher maximalen Taktfrequenz können Sie Ihr Design betreiben?

Erstellen Sie ein VHDL-Modell für eine Zählerstelle inklusive Anzeige und prüfen Sie dessen Funktion in der Simulation. Wie hoch ist der Ressourcenverbrauch in der Synthese?

Jetzt fügen Sie alle Subblöcke zu einem Gesamt-Design zusammen. Prüfen Sie die Funktion wieder via Simulation und stellen Sie den Ressourcenverbrauch in der Synthese fest.

Zur Realisierung auf dem Board benötigen Sie nun noch ein *Testbed*. Dessen Aufgabe besteht einerseits darin, Signale auf dem Board von der negativen in positive Logik zu überführen und korrekt am Spiel-Design anzuschließen. Andererseits müssen die asynchronen Signale, welche von den Tasten kommen einsynchronisiert werden, wozu die entsprechenden *units* aus der vorigen Übung verwendet werden können.
Synthetisieren Sie Ihr Modell. Welche Ressourcen werden gebraucht? Mit welcher maximalen Taktfrequenz können Sie Ihr Design betreiben?

Testen Sie die Funktion ihres Entwurfs auf dem Board.

VUnit, ein Verifikationsframework

VUnit ist ein Verifikationsframework auf der Basis von VHDL und Python. Es ist ein – sehr erfolgreiches – Open-Source-Projekt: https://vunit.github.io/index.html.

Damit stellen sich zunächst zwei Fragen: Was ist ein Framework und was ist mit Verifikation gemeint?

Ein *Framework* stellt eine Grundstruktur dar, auf deren Basis sich bestimmte Dinge deutlich leichter verwirklichen lassen. Man könnte auch von einer konkretisierten Herangehensweise (Methodik) sprechen: VUnit erleichtert das Erstellen von Verifikationsszenarien in VHDL (oder inzwischen wahlweise auch in SystemVerilog).

Mit *Verifikation* ist die automatische Überprüfung von VHDL-Modellen auf korrekte Funktion durch eine entsprechende *testbench* gemeint. Wichtig ist hierbei, dass die Überprüfung nicht etwa durch einen Menschen geschieht, z.B. mittels – mühsamen – Betrachtens der Waveform, sondern automatisch, durch die *testbench*. Das Ergebnis der Verifikation ist dann ein sehr einfaches: Entweder es passt oder es passt nicht. VUnit kann automatisch eine Vielzahl von Testfällen ablaufen lassen und gibt für jeden Testfall jeweils ein *pass* in grün oder ein *fail* in rot aus.
Der Name VUnit kürzt VHDL *unit testing* ab (VUnit basiert auf xUnit[1]). VUnit beschränkt sich aber keineswegs auf *unit tests*, sondern unterstützt jegliche Verifikationsszenarien, also auch solche, in denen mehrere oder gar alle *units* eines Systems zusammengeschaltet sind (im Sinne des Software-Qualitätsmanagements also neben den *unit tests* auch Integrations- und Systemtests).

Im Rahmen dieser LVA kann auf das systematische Erstellen automatischer *testbenches* bestenfalls am Rande eingegangen werden, aber VUnit vereinfacht auch eine debugging-orientierte Simulation mit einer einfachen *testbench* erheblich, also eine Simulation, bei der die Analyse der Waveform die zentrale Rolle spielt. Um diese Vereinfachungen soll es in dieser Übung gehen.

3. Aufgabe *Fakultativ: WORK ist keine VHDL-library!*

Die meisten Simulatoren und Synthesewerkzeuge verwenden ganz selbstverständlich den Namen WORK (work, Work) als Voreinstellung für die *library*, die für den VHDL-Code der Anwenderin oder des Anwenders vorgesehen ist.
Modelsim schlägt diesen Namen beispielsweise vor, wenn man mit dem entsprechenden Menüpunkt eine neue *library* anlegen möchte (File—New—Library...). Machen Sie bitte einen Screenshot vom entsprechenden Dialogfenster.

Mit diesem Dialogfenster kann man selbstverständlich auch eine *library* mit einem anderen Namen als work anlegen. Was ist die genaue Bedeutung der im Dialog verwendeten Begriffe *Library Name* und *Library Physical Name*? Um dies zu klären, können Sie probehalber eine *library* mit einem Phantasienamen anlegen. Anschließend untersuchen Sie bitte in der Library-Ansicht in Modelsim mittels des Kontextmenüs (rechte Maustaste) die Eigenschaften der soeben angelegten *library*.

Im Sinne des VHDL-Standards ist die Verwendung des Namens WORK für eine VHDL-*library* eine sehr problematische Vorgehensweise, weil WORK nicht etwa der Name einer bestimmten *library* ist – nicht einmal sein kann –, sondern lediglich diejenige *library* bezeichnet, in die aktuell analysiert (oder compiliert) wird. WORK ist sozusagen ein *Pointer* auf diejenige *library* in welche der derzeit zu analysierende VHDL-Code abgelegt wird, die sogenannte *current liberary*.
Daher muss man z.B. bei Modelsim den Namen der *library* angeben welche für den Compile-Vorgang (=Analysis) die Rolle von WORK einnehmen soll, z.B. mit `vcom -work WilmaLib InterfaceAdaption-e.vhd`
Es ist – leider – nicht explizit durch den Standard untersagt, dass auch eine *library* namens WORK existieren kann. Für diese lautet der Compile-Befehl in Modelsim `vcom -work work InterfaceAdaption-e.vhd`. Verwendet man den Namen WORK für eine *library*, beschwört dies aber ein Problem herauf.

Worin besteht das Problem? Wenn man innerhalb eines Modells, nennen wir es im Beispiel `InterfaceAdaption`, dass sich in einer *library*, z.B. mit dem Namen WilmaLib befindet, eine *entity* aus einer *library* namens WORK instantiieren möchte, ist dies schlicht nicht möglich!
Denn `work.Spi2AvSt(Rtl)` wird interpretiert als
`<TargetLibraryOfCurrentAnalysis>.Spi2AvSt(Rtl)` also `WilmaLib.Spi2AvSt(Rtl)`. Der

[1] https://de.wikipedia.org/wiki/XUnit

Grund: WilmaLib ist die *library* in die `InterfaceAdaption` soeben analysiert wird und damit jene *library*, welche das VHDL-LRM als *current working library* bezeichnet.
Verdeutlichen Sie den Zusammenhang anhand einer (Freihand-)Skizze, in der die Namen aus dem Beispiel verwendet werden.
Ein weiteres Beispiel ist unter
https://insights.sigasi.com/tech/work-not-vhdl-library/
zu finden.

4. Aufgabe *Fakultativ: VUnit, Installieren und Zeit sparen*

VUnit ist zum einen in Python, zum anderen in VHDL implementiert. Die Teile, die in VHDL implementiert sind, werden automatisch bei Benutzung in die entsprechenden VHDL-*libraries* compiliert. Die Teile, die in Python geschrieben sind, benötigen selbstverständlich eine Python-Installation auf dem betreffenden Rechner. Die genauen Versionsvoraussetzungen finden sich unter https://vunit.github.io/installing.html#requirements.

Mit einer aktuellen Python-Installation wird auch der Python Paket-Manager pip installiert; pip steht für *preferred installer program*. Dieser sollte zur Installation von VUnit verwendet werden: https://vunit.github.io/installing.html#using-the-python-package-manager.

Um VUnit erstmals zu verwenden, wird Ihre Beschreibung des Strobe-Generators und der dazugehörigen *testbench* aus der vorigen Übung dienen. Damit dies sinnvoll machbar ist, müssen Ihre Beschreibung und die zugehörige *testbench* selbstverständlich in der Simualation funktionieren.
Erstellen Sie nun innerhalb der *unit* `StrobeGen` zwei weitere Verzeichnnisse, nämlich `tb` (für die testbench) und `vunit` als Arbeitsverzeichnis für VUnit. Den Ordner mit ihrem synthesefähigen RTL-Modell des Strobegenerators benennen Sie um in `rtl` und verschieben ihre *testbench* in den Ordner `tb`. Im weiteren Verlauf wird die *testbench* erweitert. Machen Sie daher eine Sicherungskopie der Testbench und legen Sie diese in einen weiteren Ordner namens `archive`.

Falls Sie Scripten für die Simulation in Modelsim erstellt haben, funktionieren diese aufgrund der veränderten Pfade nicht mehr. VUnit wird einen Großteil dieser Scripte zukünftig ohnehin ersparen. VUnit erwartet, dass *testbenches* einen Dateinamen aufweisen, der mit "tb_"beginnt. Ändern Sie den Dateinamen der *testbench* (im Ordner `tb`) und ebenso deren *entity*-Namen entsprechend ab. Vergessen Sie nicht, dass auch in der Definition der *architecture* Bezug auf den *entity-Namen* genommen wird. Also auch dort bitte den Namen anpassen!

Im Ordner `vunit` benötigt VUnit ein Python-Script, dass Informationen über das Projekt enthält. Dieses Script wird dort unter dem Namen `run.py` abgelegt. Für den Strobe-Generator kann `run.py` beispielsweise so aussehen:

```
#!/usr/bin/env python3

"""
Based␣on␣VUnit's␣VHDL␣User␣Guide
---------------------------------

The␣most␣minimal␣VUnit␣VHDL␣project␣covering␣the␣basics␣of␣the
:ref:'User␣Guide␣<user_guide>'.
"""

from pathlib import Path
```

```
from vunit import VUnit

VU = VUnit.from_argv()

RTL_PATH = Path(__file__).resolve().parents[1] / "rtl"
TB_PATH = Path(__file__).resolve().parents[1] / "tb"

GRP_PATH = Path(__file__).resolve().parents[2]
#print("Grp Path:", GRP_PATH.absolute())

UNITS_PATH = Path(__file__).resolve().parents[3]
#print("Units Path:", UNITS_PATH.absolute())

lib = VU.add_library("lib")

# Packages
lib.add_source_files(UNITS_PATH / "grpPackages" / "*" / "rtl" / "*.vhd")

lib.add_source_files(RTL_PATH / "*.vhd")
lib.add_source_files(TB_PATH / "*.vhd")

VU.main()
```

Öffenen Sie nun im Ordner `vunit` nach Ihrer Wahl einen *Command Prompt* (früher als *command tool* bezeichnet) oder eine Power-Shell oder, falls git installiert ist, eine Bash-Shell. Am einfachsten geht dies innerhalb des Windows-Explorers mit der rechten Maustaste bei gedrückter Shift-Taste. Im weiteren wird dieses Eingabefenster als Shell bezeichnet. Wenn Sie das Shell-Fenster relativ stark in die Breite ziehen, erleichtert das im Weiteren die Lesbarkeit.

In der Shell geben Sie nun den Befehl `python run.py -h` ein. Es erscheint ein Text, der auch über den Befehl Auskunft erteilt, durch den er ausgegeben wurde.

Im nächsten Versuch geben Sie nun `python run.py --clean --compile` ein. Vorher finden Sie mittels des Ergebnisses des vorigen Schritts heraus, was das diese Eingabe bedeuten dürfte.
Diese Anweisung führt zu einer recht beeindruckenden Liste von Meldungen, die über die Tätigkeit von Modelsim berichten. Allerdings wird aktuell offenbar lediglich compiliert. Eine Simulation wird nicht ausgeführt. VUnit beschwert sich außerdem darüber, dass keine Tests zu finden waren.
VUnit hat im Ordner `vunit` einen Unterordner `vunit_out` angelegt, den wir näher betrachten sollten. In diesem Ordner findet sich ein Ordner namens `modelsim`, der für uns interessant ist. Die anderen Ordner enthalten die Datenbasis für VUnit und sind für uns nicht wirklich relevant. Wechseln Sie also in den Ordner `modelsim`. Offenbar verwendet VUnit eine ähnliche Ordnerstruktur, wie die für uns gewohnte: Der Ordner `modelsim` stellt das Arbeitsverzeichnis für Modelsim dar. Dort ist auch eine Datei `modelsim.ini` vorhanden. Öffnen Sie diese in einem Texteditor. Interessant sind vor allem zwei Zeilen, welche die Zuordnung der VHDL-*libraries* `vunit_lib` und `lib` zu Verzeichnissen herstellen. Beide *libraries* wurden von VUnit angelegt. Alle anderen Definitionen stammen aus der Vordefinition von `modelsim.ini`, wie sie im Installationsverzeichnis von Modelsim zu finden ist.

Offenbar hat VUnit also sowohl unser Projekt compiliert (*library* `lib`) als VHDL-Code, der zu VUnit selbst gehört (*library* `vunit_lib`). Lassen Sie uns ausprobieren, ob das Verzeichnis

`modelsim` sich wirklich wie das gewohnte Arbeitsverzeichnis für den Simulator verhält. Hierzu kopieren Sie wie üblich den Link auf Modelsim in dieses Verzeichnis und öffnen Modelsim anschließend damit. Da Modelsim das Verzeichnis als Arbeitsverzeichnis betrachtet, wird die vorhandene Datei `modelsim.ini` als Initialisierungsdatei verwendet. Die von VUnit angelegten *libraries* sind also für Simulationen verfügbar. In der *library* `lib` finden Sie den Strobe-Generator mit seiner *testbench*. Simulieren Sie ihn nun. Es sollte alles wie gewohnt funktionieren. Nur eben nicht aus der *library* `work`, sondern aus `lib`.

In diesem Schritt hat VUnit bereits alle Informationen aus `run.py` ausgewertet. Suchen Sie nach der (einen!) Stelle, wo der Name für die *library* Ihres Modells festgelegt wird und ändern Sie diesen ab. Führen Sie nun wieder `python run.py --clean --compile` aus.
Versuchen Sie probehalber auch den Namen `work`, wie er von Modelsim üblicherweise verwendet wird. Was passiert?

Es ist wichtig, dass Sie den Inhalt der Datei `run.py` verstehen, um VUnit sinnvoll anwenden zu können. Analysieren Sie das Beispiel mit Hilfe der Internet-Suche[2]:

- Was macht der Aufruf `VU = VUnit.from_argv()`
- Was ist `pathlib` und was wird von dort importiert?
- Was ist das Ergebnis von `Path(__file__)`?
- Wozu dient `resolve()`?
- Wozu dient `parents[2]`?
- Was macht der Befehl `add_library("lib")`?
- Wozu dient der Befehl `add_source_files`?
- Was macht die abschließende Methode `VU.main()`?

Beschreiben Sie den Nutzen, der sich auf der bisher erreichten Stufe durch VUnit ergibt.

5. Aufgabe *Fakultativ: VUnit, Single Test*

Ergänzen Sie Ihre *testbench* für den Strobe-Generator entsprechend folgenden Beispiels[3] mit den gekennzeichneten Zeilen:

```
-- Source: https://vunit.github.io/user_guide.html

-- Add the following library clause and context clause!
library vunit_lib;
context vunit_lib.vunit_context;

entity tb_example is
  -- Add this generic!
  generic (runner_cfg : string);
end entity;

architecture tb of tb_example is
```

[2] Beachten Sie, das Python eine objektorientierte Sprache ist.
[3] Weitere Erläuterungen finden Sie unter https://vunit.github.io/user_guide.html

```
begin
  main : process
  begin
    -- Your testbench needs to begin with a call to test_runner_setup
    test_runner_setup(runner, runner_cfg);

    -- This is the part where your original testbench code belongs
    -- ...
    report "Hello world!";
    -- ...
    -- End of the original testbench code.

    -- Your testbench needs to finish with a call to test_runner_cleanup
    test_runner_cleanup(runner);
  end process;
end architecture;
```

Beachten Sie dabei, dass die *procedure calls* nicht innerhalb eines *process* liegen müssen, jedenfalls aber am zeitlichen Anfang und am Ende der geplanten Simulation.
Eine Möglichkeit ist es, diese aus einem eigenen *process* aufzurufen, der dann auch die Dauer der Simulation bestimmt:

```
  WaitLongEnough : process is
  begin
    test_runner_setup(runner, runner_cfg);
    wait for 10 us;
    test_runner_cleanup(runner);
  end process;
```

Ist dieser *process* Teil der *testbench*, dauert die Simulation also 10 us lang.

VHDL-Code, der in Ihrer *testbench* bislang dazu diente, die Simulation zu beenden (`assert false...`), kommentieren Sie bitte aus.

Nun weisen Sie VUnit mittels `python run.py` zur Ausführung aller Tests an. In ihrem Beispiel gibt es nur einen Test, der den Namen der *testbench-entity* besitzt. VUnit wird nun automatisch die notwendigen VHDL-Dateien neu compilieren und anschließend die Simulation aller Tests starten. Da Sie in der *testbench* nur einen Test implementiert haben, wird auch nur ein Test ausgeführ. Dieser sollte fehlerfrei ausgeführt werden, was durch entsprechende Nachrichten von VUnit quittiert wird.

Provozieren Sie nun in der *testbench* einen Fehler, der mittels Assert-Anweisung (*severity error*) detektiert wird. Führen Sie wieder mittels `python run.py` alle Tests durch. Es gibt nach wie vor nur einen Test, nämlich den mit dem Namen der *testbench-entity*. Dieser sollte nun den Fehler anzeigen und die Simulation zum Zeitpunkt des Fehlerauftritts stoppen.

Im Fehlerfall werden Sie die Simulator-GUI zwecks Debugging zuhilfe nehmen wollen. Daher rufen Sie nun alle Tests (da nach wie vor nur einer vorhanden!) in der Simulator-GUI auf: `python run.py --gui`. Dort müssen Sie die Simulation eigens von der TCL-Shell aus mit `vunit_run` starten.
Im Simulator sind durch VUnit neben dem Befehl `vunit_run` einige Befehle definiert: https://vunit.github.io/cli.html#opening-a-test-case-in-simulator-gui.

Unter anderem können Sie VUnit mittels `vunit_restart` anweisen, alle erforderlichen VHDL-Dateien nach Ihren Änderungen neu zu compilieren und die Simulation wieder von vorn zu starten.
Wenn Sie die Konfiguration des Waveform-Fensters für zukünftige Simulationsläufe speichern wollen (Datei `wave.do`), können Sie das auf dem gewohnten Weg tun: (`File|Save Format...`).

Finden Sie in der Simulation heraus, was der Inhalt des *generics* `runner_cfg` ist.

VUnit unterstützt die Unterbringung mehrerer Tests in einer einzigen *testbench*. Wie müssten Sie hierzu Ihre *testbench* erweitern? Hinweise finden Sie unter https://vunit.github.io/user_guide.html#vhdl-test-benches

6. Aufgabe *Fakultativ: Code-Tastatur*

Im Rahmen dieser Übung sollen Sie die Unterschiede zwischen einer Mealy- und Moore-Maschine in der Praxis nachvollziehen.

Eine Anwendung von endlichen Automaten stellt die Erkennung einer bestimmten Zahlenfolge dar. Eine Code-Tastatur arbeitet nach dem selben Prinzip, wobei die Reihenfolge eine große Rolle spielt – bei falscher Eingabe muss die Zahlenfolge neu von vorn eingegeben werden. Der Zahlenbereich bewegt sich zwischen 0 und 9. Die Zahlenfolge unserer Code-Tastatur lautet **9 - 3 - 4 - 6** (1. Zahl = 9, 2. Zahl = 3, usw.). Solange die Zahlenfolge falsch ist (also noch nicht korrekt eingegeben wurde), ist das Zahlenschloß verschlossen. Der endliche Automat zeigt dies durch einen aktiven Ausgang `oLocked` an. Der Ausgang wird inaktiv gesetzt, sobald eine korrekte Eingabe getätigt worden ist und bleibt solange inaktiv bis die nächste Zahl eingegeben wird.

Bevor Sie mit der Implementierung beginnen, sollten Sie die beschriebene Funktion in eine Zustandsmaschine (endlicher Automat) umsetzen. Die Beschreibung dieser Zustandsmaschine erfolgt in dieser Aufgabe (und auch sehr häufig in der Praxis) mittels Bubble-Diagramm (Zustandsgraph oder Zustandsdiagramm).

Abbildung 1 zeigt ein Beispiel (also nicht etwa die FSM des Schlosses) für ein Bubble-Diagramm mit den drei Zuständen A, B und C, dem Input X und dem Output Y. Die Zustandsüberführung von Zustand A auf Zustand B erfolgt, wenn der Input X den Wert '1' aufweist. Der Wert von Input X spielt bei der Zustandsüberführung von Zustand B auf Zustand C keine Rolle. Der Output Y ist nur vom aktuellen Zustand abhängig (Moore-Maschine). Bei einer Mealy-Maschine wird der Output bei der Zustandsüberführung (vgl. Abbildung 2) angeben, da er vom aktuellen Zustand und den Eingangswerten abhängt.

Erstellen Sie zunächst für Ihre Zahlenschloß-Zustandsmaschine nach Moore ein Bubble-Diagramm. Tragen Sie alle Zustandsübergänge ein!

Anschließend erstellen Sie die Zustandsmaschine nach Mealy in Form eines zweiten Bubble-Diagramms. Achten Sie hier ebenfalls auf die korrekte Beschriftung aller Zustandsübergänge.

Implementieren Sie nun den Zustandsautomaten nach Moore in VHDL (*architecture* `CombinationLockMoore`). Verwenden Sie für die Zustandsmaschine zwei Prozesse!

Weisen Sie die Funktion mittels Simulation nach.

Synthetisieren Sie Ihr Modell. Notieren Sie die Anzahl der Flip-Flops. Welche *Warnings*, *Critical Warnings* und *Errors* erhalten Sie?

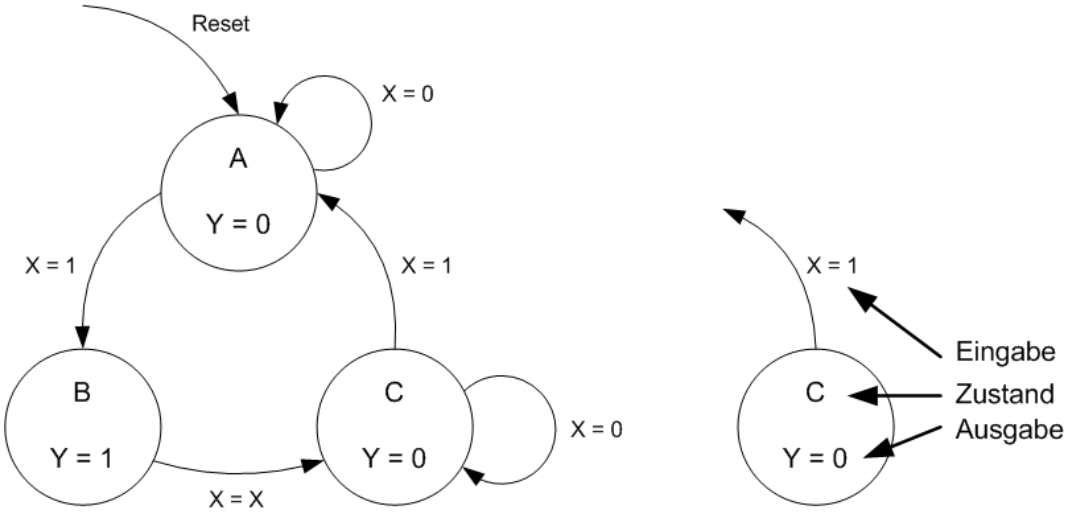

Abbildung 1: *Bubble*-Diagramm einer Zustandsmaschine nach *Moore*.

Realisieren Sie den Entwurf auf dem DE1-Board. Als Taktquelle nutzen Sie den Einzelschritttaster. Die Zahlen 0 bis 9 werden durch die Schiebeschalter umgesetzt (z.B. Switch `SW0` auf *On* entspricht der Eingabe von Zahl 0 usw.). Wird kein Schiebeschalter auf *On* gesetzt, dann verharrt die FSM in ihrem aktuellen Zustand. Die Ausgabe der Zustände und des Locked-Ausgangs erfolgt über die verügbaren LEDs. Mit welcher maximalen Taktfrequenz können Sie Ihr Design betreiben?

Implementieren Sie nun auch die Mealy-Zustandsmaschine in VHDL (z.B. *architecture* `CombinationLockMealy`). Behalten Sie die Zwei-Prozeß-Methode bei.

Weisen Sie die Funktion ebenfalls mittels Simulation nach.

Synthetisieren Sie Ihr Modell. Notieren Sie die Anzahl der FlipFlops.

Realisieren Sie den Entwurf auf dem Board. Mit welcher maximalen Taktfrequenz können Sie dieses Design betreiben? Welche *Warnings*, *Critical Errors* und *Errors* erhalten Sie? Welche FPGA-Pins werden von Ihrem Design verwendet?

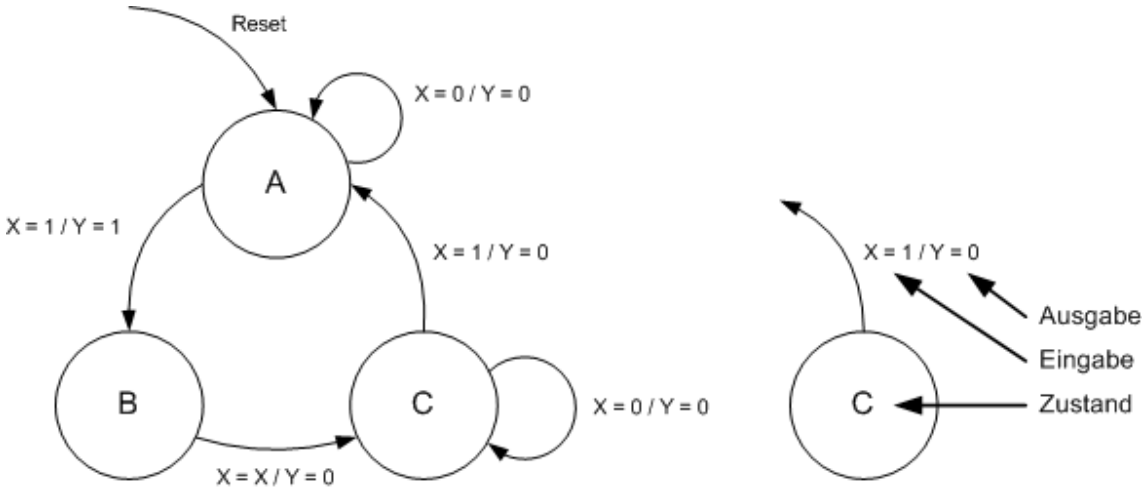

Abbildung 2: *Bubble*-Diagramm einer Zustandsmaschine nach *Mealy*