1. VHDL
1.1. Entities

Grundaufbau einer Entity

```
entity entity_name is
    port(
        -- IO Deklaration
    );
end entity_name;
```

Beispiel Entity eines 4-Fach Multiplexers

```
entity MUX4 is
    port(
        S: in bit_vector(1 downto 0);
        E: in bit_vector(3 downto 0);
        Y: out bit;
    );
end MUX4;
```

1.2. Architectures

Grundaufbau einer Architecture

```
architecture arch_name of entity_name is
    -- Architecture Deklaration
    begin
        -- VHDL Anweisungen
end arch_name;
```

Beispiel Architecture eines 4-Fach Multiplexers.

Bei Vektoren lassen sich auch die einzelnen Stellen direkt setzen!

```
architecture BEHAV of MUX4 is
    begin
        with S select
            Y<= E(0) when "00",
```

```
                E(1) when "01",
                E(2) when "10",
                E(3) when others; -- bzw. when '11'
end BEHAV;
```

1.3. Datentypen

Datentyp	Zulässige Werte
bit, bit_vector	‘0’, ‘1’
std_ulogic, std_ulogic_vector	‘U’, ‘X’, ‘0’, ‘1’, ‘w’, ‘L’, ‘H’, ‘-’
std_logic, std_logic_vector	‘U’, ‘X’, ‘0’, ‘1’, ‘w’, ‘L’, ‘H’, ‘-’
integer	-21478364 bis 2147483647
natural	0 bis 2147483647
positive	1 bis 2147483647

1.4. Libraries

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_ulogic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL; -- Für arithmetische Operationen auf std_logic_vector.
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Die IEEE Library bietet die oben genannten std_logic und std_ulogic Datentypen.

1.5. Selektive & Bedingte Abfragen

1.5.1 Selektiv

Es müssen **alle** möglichen Werte des Steuersignals in expliziten when-Zweigen berücksichtigt werden (alterantiv auch „when others“-Zweig)

```
architecture BEHAV of MUX4 is
    begin
        with S select
            Y<= E(0) when "00",
                E(1) when "01",
                E(2) when "10",
```

```
                E(3) when others; -- bzw. when '11'
end BEHAV;
```

1.5.1 Bedingt

```
architecture BEHAV of MUX4 is
    begin
        Y<= E(0) when S="00" else
            E(1) when S="01" else
            E(2) when S="10" else
            E(3);
end BEHAV;
```

1.6. Process

!!! Wichtigstes VHDL-Syntaxkonstrukt !!!

- Prozesse werden ausgeführt, wenn sich ein Signalwert in der Empfindlichkeitsliste ändert (event)
- Alle Prozesse in einer architecture werden **nebenläufig/gleichzeitig** ausgeführt!
- Innerhalb eines Prozesses werden Anweisungen **sequentiell** ausgeführt!
- Signalwertänderung erfolgt immer erst am Prozessende
- Geänderte Werte innerhalb der Sequenz = Variablen
- Wenn Variablen nach Prozess noch verwendet werden: Variable -> Arch Signal
- Austausch zwischen Prozessen erfolgt mit den lokalen Signalen der architecture (bidirektionale Kommunikation 2 Signale!!!)
- Innerhalb von Prozessen sind nur **sequentielle Anweisungen** (case)! (NICHT: with select und when else)

Grundaufbau eines Process use ieee.std_logic_1164.all; use ieee.std_ulogic_1164.all; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_UNSIGNED.ALL;

```
<Label>: process (<Empfindliche Signale>)
    -- Deklaration
    begin
        -- Sequentielle Anweisungen
end process <Label>;
```

Beispiel eines Process

```
entity MUX4 is
    port(
        S: in bit_vector (1 downto 0);
        E: in bit_vector (3 downto 0);
        Y: out bit;
    );
end MUX4;

architecture BEHAV of MUX4 is
    begin
        MUXPROC: process(S,E)
            begin
                case S is
                    when '00' => Y <= E(0);
                    when '01' => Y <= E(1);
                    when '10" => Y <= E(2);
                    when '11" => Y <= E(3);
                end case;
        end process MUXPROC;
end BEHAV;
```

In der **Empfindlichkeitsliste** stehen alle Signale, die auf der **rechten Seite** einer Zuweisung oder in einer **Abfrage** stehen!

1.7. Components

Grundaufbau eines Component

```
singal -- Alle in & out Signale der entities
component entity_name
    port(
        -- IO Deklaration wie in entity
    );
end component;
```

Beispiel eines Volladdierers

```
entity FULLADD is
    port(
        E0, E1, CIN: in bit;
        SUM, COUT: out bit;
```

```
    );
end FULLADD;

architecture BEHAV of FULLADD is
    signal SUM1, SUM2, COUT1, COUT2: bit;
    component HALFADD
        port(
            E0, E1: in bit;
            SUM, COUT: out bit;
        );
    end component;
        begin
            U1: HALFADD port map(E0 => E0, E1 => E1, SUM =>SUM1, COUT => COUT1);
            U2: HALFADD port map(E0 => SUM1, E1 => CIN, SUM => SUM2, COUT => COUT2);
            COUT <= COUT1 or COUT2;
            SUM <= SUM2;
end BEHAV;
```

2. Synchrone (sequentielle) Logik

2.1. Finite State Machines

2.1.1. Moore

- Ausänge hängen nur von den Zuständen ab
- Ausänge stehen in den States

Bei einer Moore FSM hat man in folgendem Beispiel 3 States

Man benötigt immer 2 Tabellen!

Einmal State transition table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Und einmal Output table

Current State S	Output Y
S0	0
S1	0
S2	1

2.1.2. Mealy

- Ausgänge hängen sowohl von den **Zuständen**, als auch von den **Eingängen** ab
- Ausänge werden auf die Pfeile zwischen den States geschrieben
- Pfeile haben dabei Syntax wenn Input A dann Output Y (A/Y)

Bei einer Mealy FSM braucht man hier nur 2 States und die beiden Tabellen sind kombiniert

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

2.2. FSM to Schematic

2.2.1. State transition table

Die state transition table gibt an, für welchen state (S) und welche Eingangssinale sich welcher next state (S') ergibt. 'X' bedeutet 'Dont care'

Beispiel an einer Ampelschaltung

Current State S	Input A	Input B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Wenn im State S0 auf In1 false kommt, geht es egal was In2 macht, in den State S1.

Danach werden die States binär kodiert

State	Encoding
S0	00
S1	01
S2	10
S3	11

Hier immer **alle** Möglichleiten anlegen! Also wenn es nur 3 States (00, 01, 10) sind, trotzdem noch State 11 anlegen! Der Tabelleneintrag dazu sieht folgendermaßen aus:

Current S1	Current S0	Input A	Input B	Next S1'	Next S0'
1	1	X	X	X	X

Dann die beiden kombinieren

Current S1	Current S0	Input A	Input B	Next S1'	Next S0'
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Mit hilfe eines KV Diagramms kann man die Boolsche Gleichung herrausfinden.

S1' und S0' werden als separate 4x4 KV-Diagramme behandelt!

Der 'Dont Care' Zustand X wird mit im Diagramm berücksichtigt, um sie mit 1en zu Verbinden! Ein Verbund aus reinen X wird **nicht** berücksichtigt!

In diesem Beispiel ist es:

S1' = S1 **XOR** S0

S0' = !S1 !S0 !InputA **OR** S1 !S0 !InputB

2.2.2. Output table

Jeder State hat einen definierten Output.

Im Fall der Ampelanlage, gibt es 2 Ampel (waagrechte LA und senkrechte LB) und 3 Werte (green, yellow, red).

Erst werden die Zustände binär kodiert

Output	Encoding
green	00
yellow	01
red	10

Dann mit den State anhand des Graphen kombinieren

Current S1	Current S0	LA1	LA0	LB1	LB0
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Mit hilfe eines KV Diagramms kann man wieder die Boolsche Gleichung herrausfinden.

LA1, LA0, LB1, LB0 werden als separate 2x2 KV-Diagramme behandelt!

Der ‘Dont Care’ Zustand X wird mit im Diagramm berücksichtigt!

In diesem Beispiel ist es:

LA1 = S1

LA0 = !S1 S0

LB1 = !S1

LB0 = S1 S0

2.2.3. Schematic

Inputs -> Next state logic -> Register mit CLK -> Output logic -> Outputs

2.2.4. VHDL

```
entity MODULE is
    port(
        -- TODO
    );
end MODULE;
```

3. FSM in VHDL

Ähnlich vorgehen, wie bei der Erstellung des Schematics!

1. Register Process erstellen
2. FSM Process (Next State und Output Logic) erstellen

3.1. FSM als Entitiy erstellen

```
entity FSM is
    Port (
        --CLK, RESET & ENABLE immer als bit oder std_logic
        CLK     : in std_logic;
        RESET   : in std_logic;
        INPUT   : in std_logic_vector(1 downto 0);
        OUTPUT  : out std_logic_vector(1 downto 0)
    );
end FSM;
```

3.2. Definieren der Architecture

```
architecture Behavioral of FSM is
    type state_type is (S0, S1, S2, S3); -- erstellt Datentyp 'type' mit den Namen
    ↪  'state_type'
    signal state, next_state: state_type; -- erstellt lokale Signale mit den Namen
    ↪  'state' und 'next_state' vom vorher generierten 'state_type'
begin
```

3.3. Zustandspeicher

```
MEM:process(CLK, RESET)
begin
    if RESET = '1' then
        state <= S0;
    elsif rising_edge(CLK) then
        state <= next_state;
    end if;
end process;
```

3.3.1. Moderlierung der Taktflanken

```
CLK='1' and CLK'event --(steigend)
-- mit std_logic auch:
rising_edge(clk)
```

```
CLK='0' and CLK'event --(fallend)
-- mit std_logic auch:
falling_edge(clk)
```

- `RESET` : Asynchroner Reset in if vor Taktflanke
- `ENABLE` : if unterhalb Taktflanke

3.4. Next-State Logic und Output Logic

Im FSM Prozess wird der Wert des lokalen Signals `next_state` auf den Wert des neuen Zustands gesetzt.

Bei der nächsten rising edge der CLK, wird der Wert von `next_state` auf `state` übertragen!

```
-- MOORE FSM
-- Output ist nur abhängig vom state
-- Erst wird überprüft, in welchem state man ist, auf Basis des state wird der output
↪  gesetzt, dann auf Basis des inputs der next_state gesetzt

MOORE:process(state, INPUT) -- Wird ausgeführt, sobald sich state oder INPUT ändert
begin

    -- Default Values
    next_state <= state; -- Default next_state ist aktueller state

    case state is
        when S0 =>                  -- Wenn state == S0
            OUTPUT <= '00';         -- Setze OUTPUT auf '00'
            if INPUT = '01' then    -- Wenn INPUT '01'
                next_state <= S1;   -- next_state == S1
            else                    -- Wenn INPUT nicht '01'
                next_state <= S0;   -- next_state == S0
            end if;
        when S1 =>
            OUTPUT <= '01';
            if INPUT = '10' then
                next_state <= S2;
            else
                next_state <= S1;
            end if;
```

```
        -- Weitere Zustände
        when others =>
            next_state <= S0;
    end case;
end process;
```

```
-- MEALY FSM
-- Output ist abhängig von state und Input
-- Erst wird überprüft, in welchem state man ist, auf Basis des Inputs werden output
↪  und next_state gesetzt

MEALY:process(state, INPUT)
begin
    case state is
        when S0 =>                  -- Wenn state == S0
            if INPUT = '01' then    -- Wenn INPUT == '01'
                OUTPUT <= '11';     -- Setze OUTPUT auf '11'
                next_state <= S1;   -- next_state == S1
            else                    -- Wenn INPUT nicht '01'
                OUTPUT <= '00';     -- Setze OUTPUT auf '00'
                next_state <= S0;   -- next_state == S0
            end if;
        when S1 =>
            if INPUT = '10' then
                OUTPUT <= '10';
                next_state <= S2;
            else
                OUTPUT <= '01';
                next_state <= S1;
            end if;
        -- Weitere Zustände
        when others =>
            next_state <= S0;
    end case;
end process;
```

4. Timing

4.1. Wichtige Zeiten

- Setup Time `Tsetup` : Zeit vor der Taktflanke, in der das Eingangssignal stabil sein muss

- Hold Time `Thold` : Zeit nach der Taktflanke, in der das Eingangssignal stabil bleiben muss
- Clock-to-Q Propagation Delay `Tpcq` : Zeit bis der Ausgang stabil ist
- Clock-to-Q Contamination Delay `Tccq` : Zeit, ab wann sich der Ausgang nach einer Eingangsänderung ändert, aber noch nicht stabil ist

4.2. System Timing

- Taktfrequenz `Fc` : Umkehrwert der Taktperiode `Tc`
- Setup Time Constraint: `Tc` >= `Tpcq` + `Tpd` + `Tsetup`
- Hold Time Constraint: `Tccq` + `Tcd` >= `Thold`

4.3. Metastabilität

- Ein FlipFlop kann metastabil werden, wenn das Eingangssignal während der `Aperture Time` nicht stabil ist
- Ein Synchronisierer (Synchronizer) minimiert die Wahrscheinlichkeit metastabiler Zustände, indem er asynchrone Signale synchronisiert

4.4 Pipelining

4.4.1. Latency und Throughput

- Latenz: Zeit, die benötigt wird, um einen Token durch das System zu schleusen
- Durchsatz: Anzahl der Tokens, die pro Zeiteinheit verarbeitet werden können

4.4.2. Paralleler Betrieb

- Spatial Parallelism: Mehrere Hardware-Kopien bearbeiten Aufgaben gleichzeitig.
- Temporal Parallelism: Aufgaben werden in kleinere Teilschritte zerlegt (Pipelining).

4.2. Maximal Taktfrequenz

Gegeben:

- `Tpcq` (Clock-to-Q Propagation Delay) = 80ps
- `Tsetup` (Setup Time) = 50ps
- `Tpd` (Propagationsverzögerung eines Gatters) = 40ps

Formel:

Setup Time Constraint: `Tc` >= `Tpcq` + `Tpd` + `Tsetup`

Berechnung:

`Tc` = `Tpcq` + `Tpd` + `Tsetup`

`Tc` = 80ps + 40ps + 50ps

`Tc` = 170ps

Maximale Taktfrequenz `Fmax` :

`Fmax` = 1 / Tc `Fmax` = 1 / 170ps `Fmax` = 1 / 170 * 10^-12 `Fmax` ≈ 5.88 GHz

4.3. Hold-Bedingung

`Tccq` + `Tcd` >= `Thold`