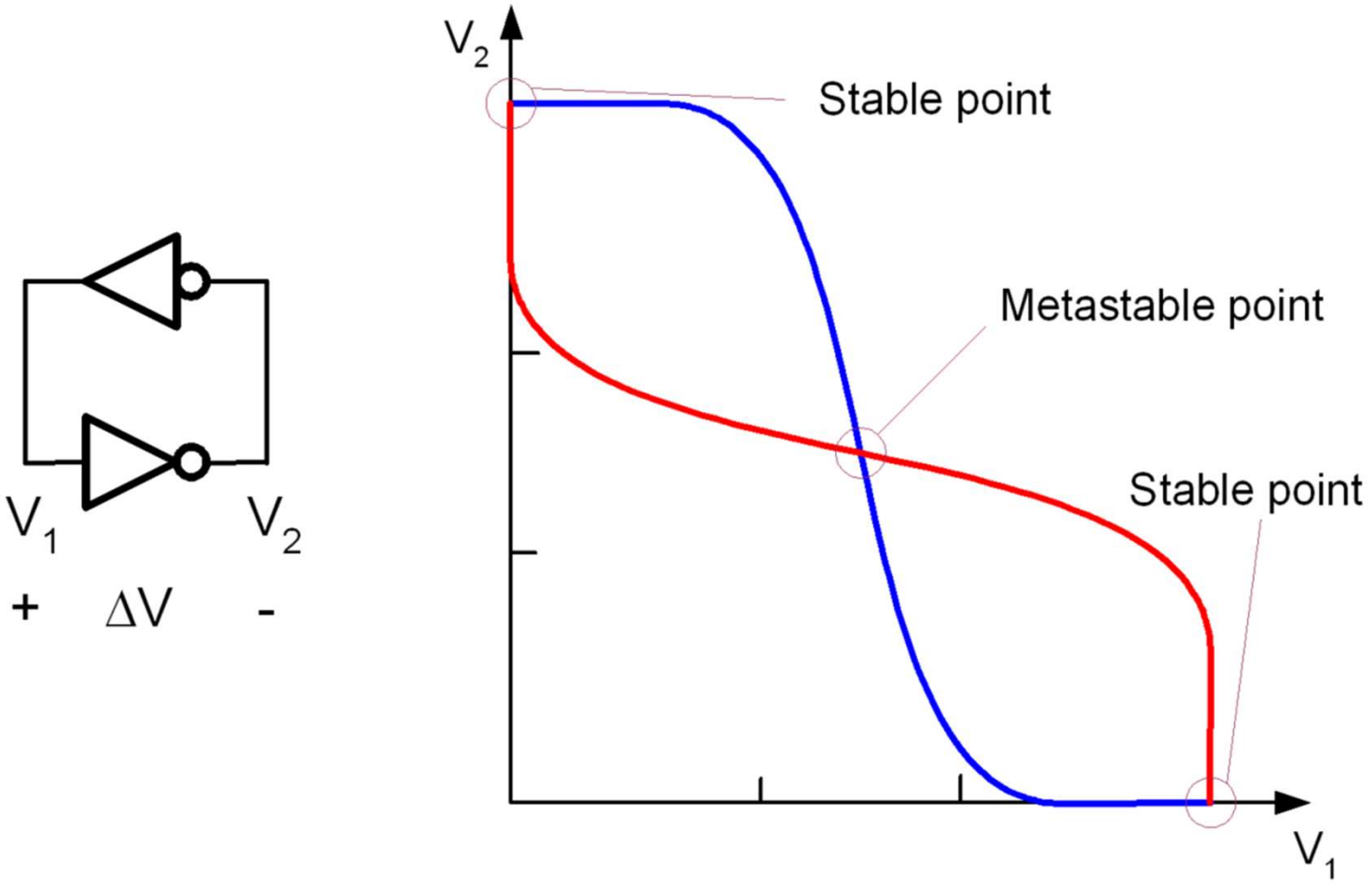
Dynamik am DFF

- **3 wesentliche Zeiten am DFF**
  - alle Zeiten bezogen auf die aktive Taktflanke T
  - t_{pd} Verzögerung T -> Q (analog tpd Gatter)
  - t_s Setup: Mindestzeit, in der D vor T stabil bleiben muss
  - t_h Hold: Mindestzeit, in der D nach T stabil bleiben muss

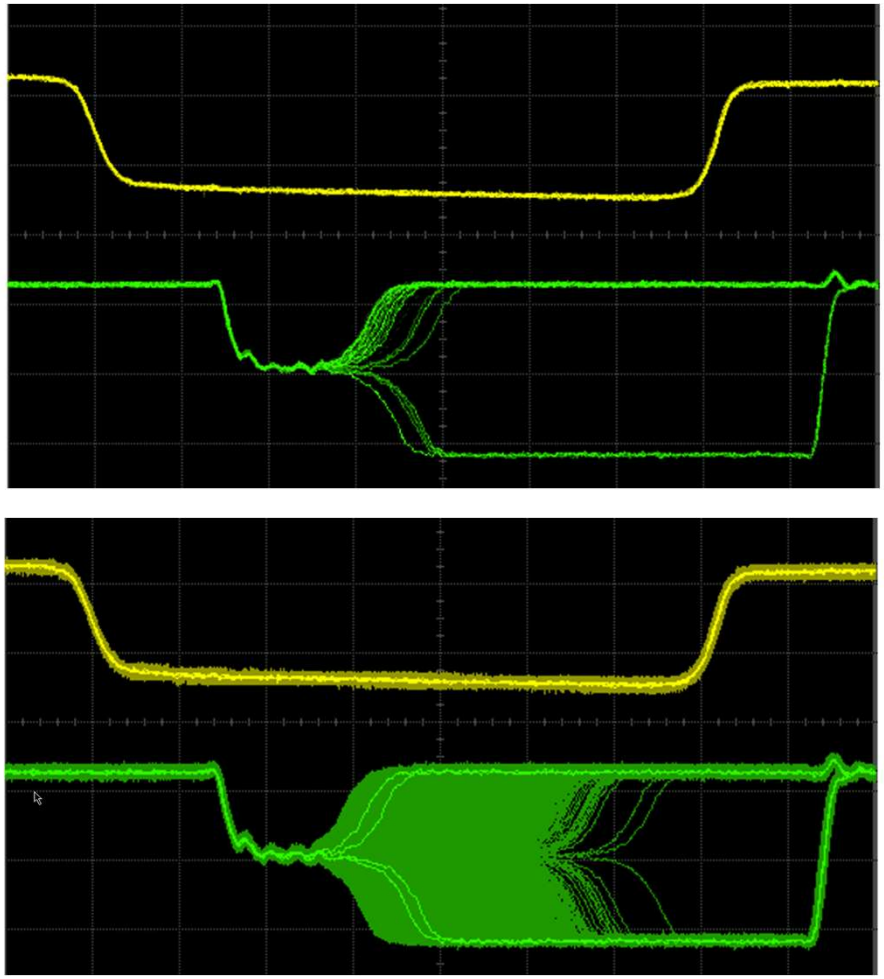
- **Metastabiler Zustand**
  - Ursache ist Verletzung von t_s oder t_h
  - Q hat für unbestimmte Zeit einen unbestimmten Wert

Entstehung des metastabilen Zustands



Quelle:
https://dokumen.tips/documents/11192005ee-108a-lecture-13-c-2005-w-j-dally-ee108a-lecture-13-metastability.html

Wahrscheinlichkeitsverteilung (Zeit)



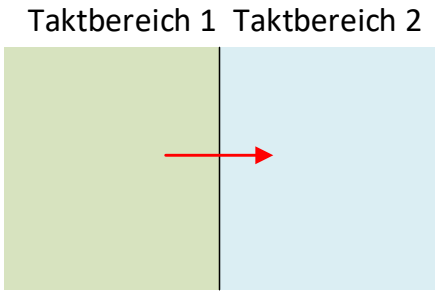
Quelle:
https://dokumen.tips/documents/11192005ee-108a-lecture-13-c-2005-w-j-dally-ee108a-lecture-13-metastability.html

Synchrones Design

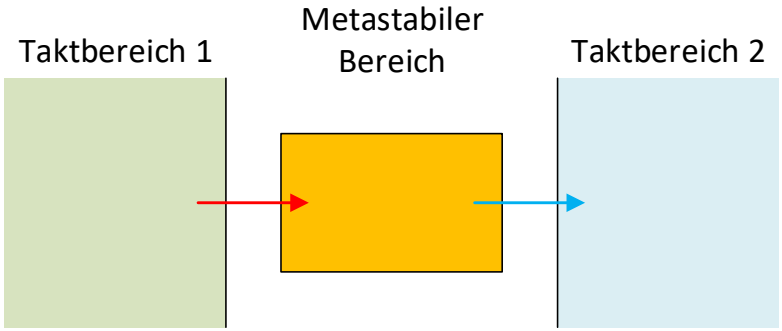
- **(möglichst) alle DFF schalten mit demselben Takt**
  - Eine gemeinsame Taktleitung (im FPGA vorhanden)
  - Unterschiedliche Frequenzen durch Taktfreigaben
  - Hohe Impulsströme möglich (Elemente schalten zum gleichen Zeitpunkt

- **Sofern nicht möglich:**
  - Größere Bereiche mit gleichen Takt bilden
  - Übergänge von einem Bereich zum anderen kann Einsynchronisieren von Signalen erfordern

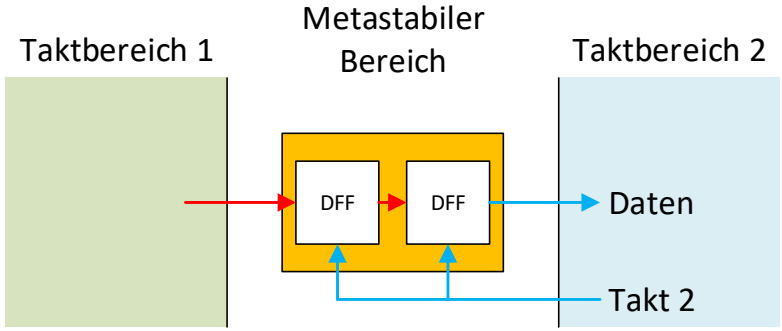
Übergang zwischen Taktbereichen



Taktbereich 1
kann auch extern sein



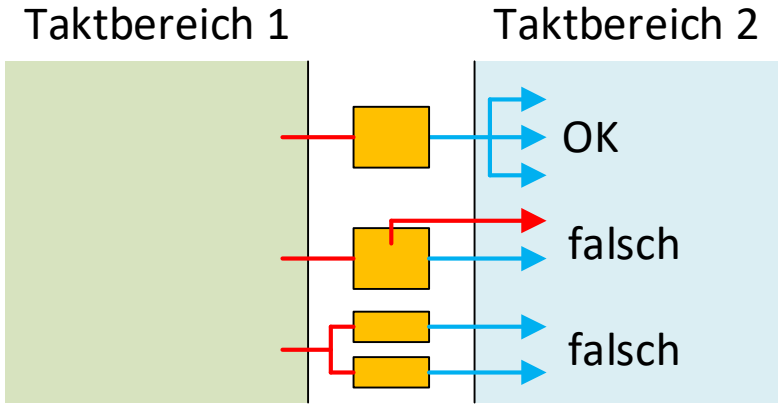
Einfügen eines
Synchronisierers



Typische DFF-Kette

Anzahl der Stufen abhängig von
- Takt 2
- Technologie der DFF

Anwendung für ein Signal



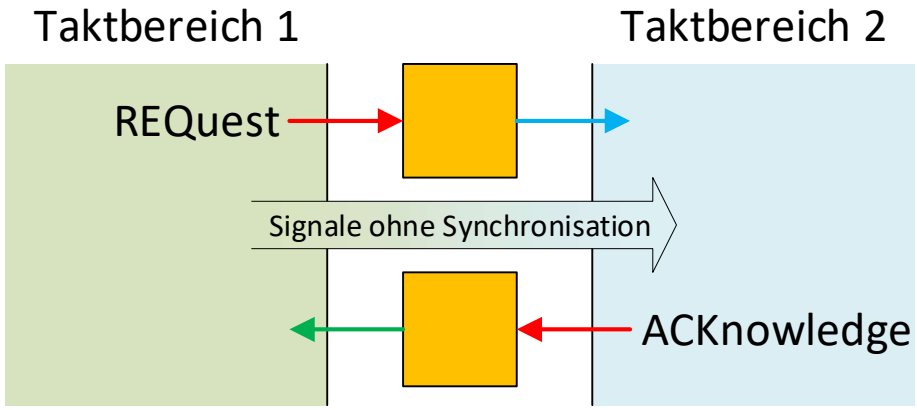
Nicht auf Signale im Synchronisierer zugreifen

Ein Signal nur **einmal** in einen Taktbereich einsynchronisieren

Anwendung für mehrere Signale

Handshake

- Übergabe mehrerer Signale (Bus)
- Übergang zwischen Taktbereichen unbekannter Frequenzverhältnisse



Request (REQ)

- Anforderung T1->T2

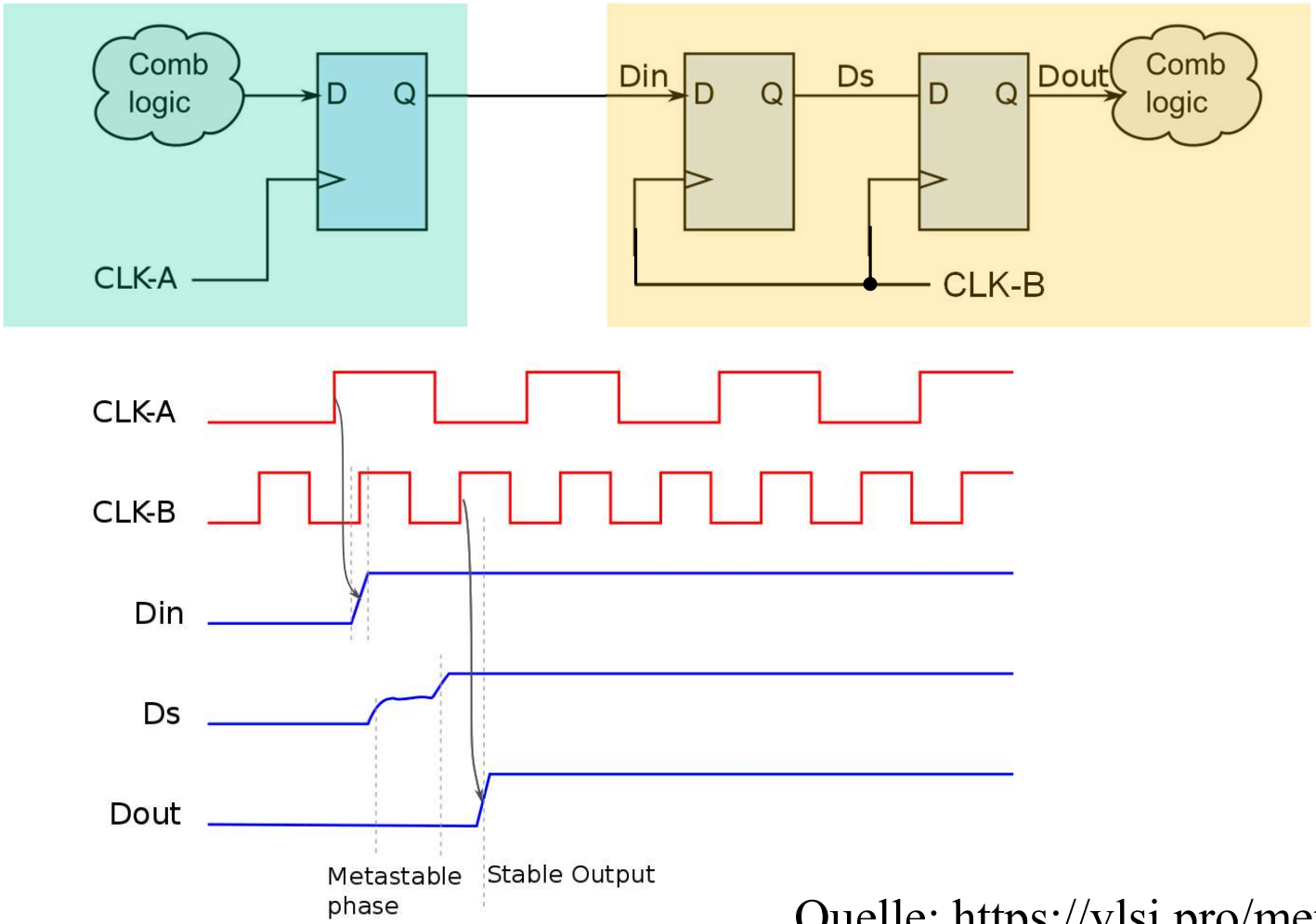
Acknowledge (ACK)

- Bestätigung T2 -> T1

Signale ohne Synchronisation

- müssen ab REQ bis ACK stabil bleiben (Aufgabe für Taktbereich 1)

Übergang zwischen Taktbereichen

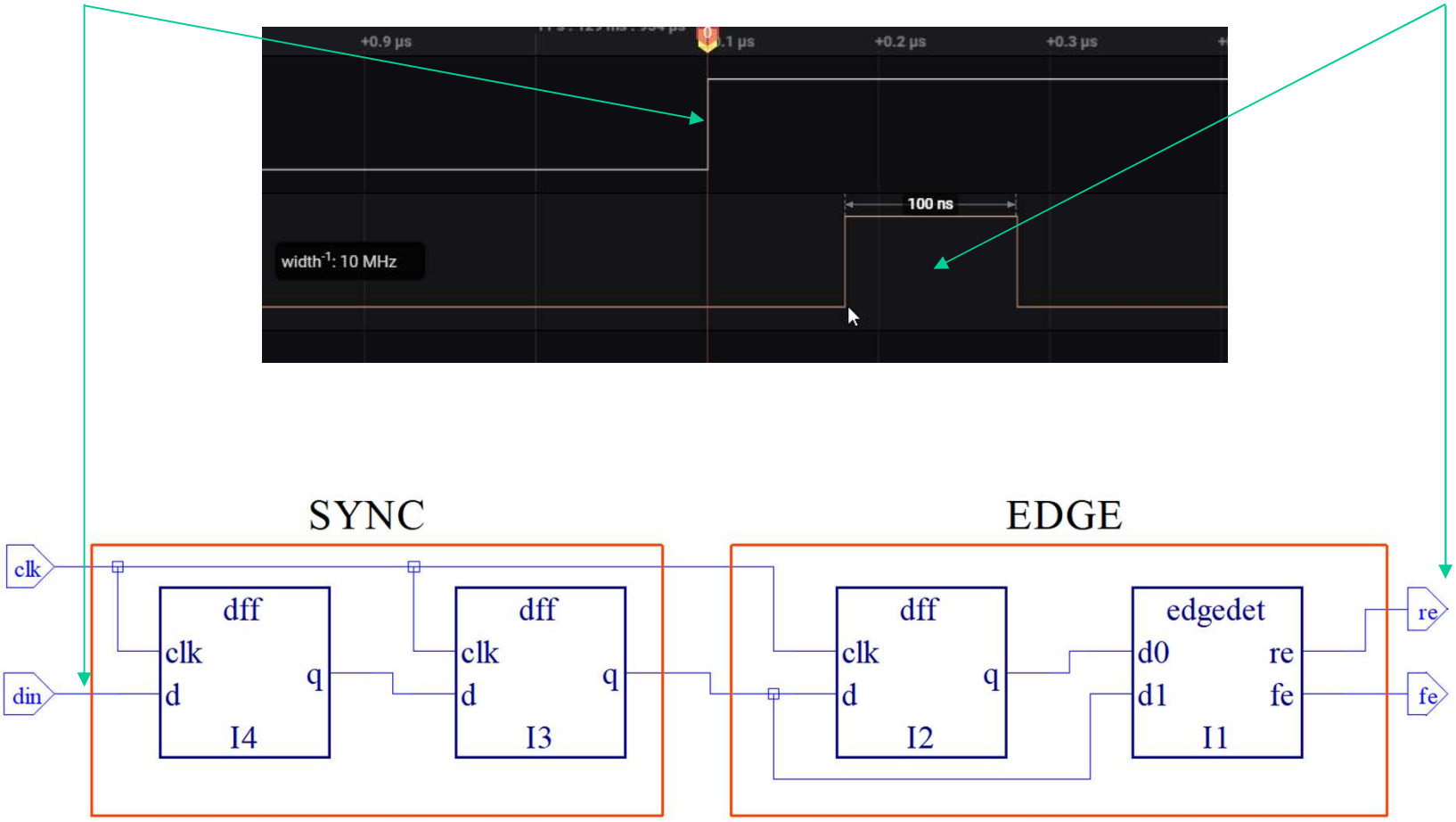


Quelle: https://vlsi.pro/metastability/

Flankenerkennung

Asynchrones Taktsignal

Synchronisiertes Enable-Signal



Automat

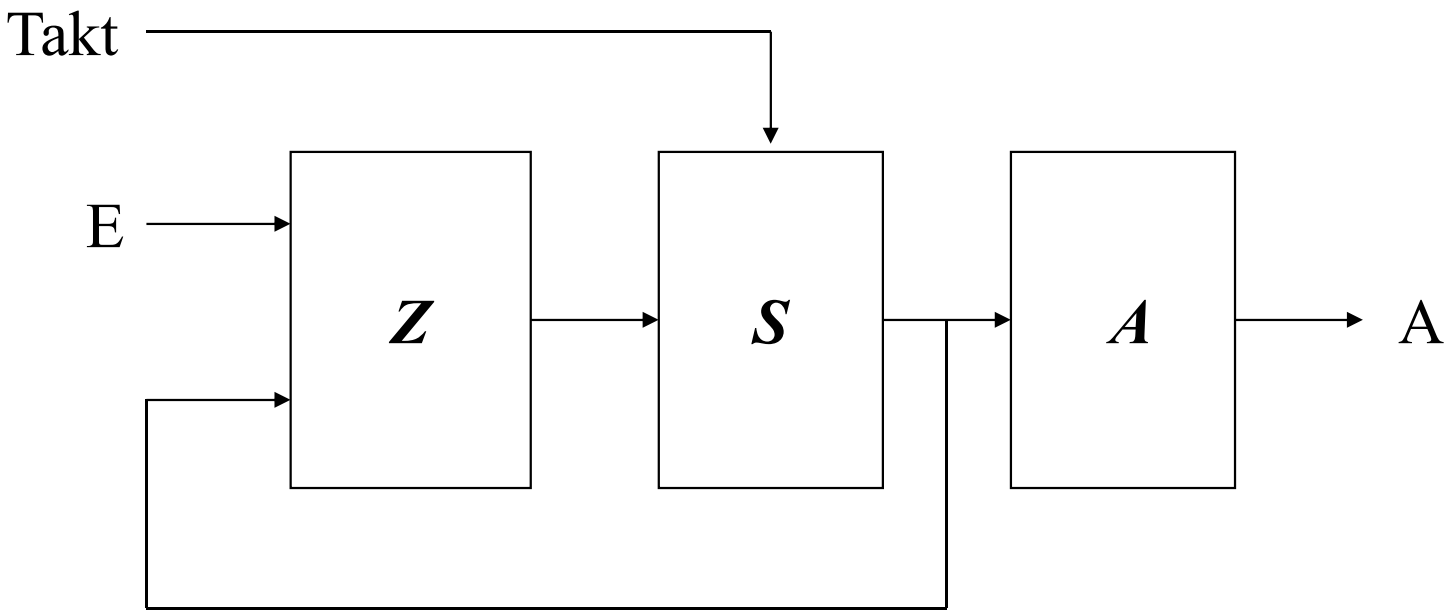
- **Schaltung mit Gedächtnis**
  - Gedächtnis wird durch Speicher (Flipflops) realisiert
  - Endliche Anzahl möglicher verschiedener Zustände
- **Automat durchläuft Zustände**
  - Zustandswechsel zeitlich bestimmt durch Taktsignal
  - Folgezustand bestimmt durch
    - Aktuellen Zustand
    - Eingangssignale zum Zeitpunkt des Zustandswechsels
- **Zwei mögliche Ausgabefunktionen**
  - nur abhängig vom Zustand (Moore)
  - abhängig von Zustand und Eingangssignale (Mealy)

Automatenentwurf

1. **Festlegen des gewünschten Verhaltens**
   - State Diagram
2. **Zustandskodierung**
   - Zuordnung von Zuständen zu Dualzahlen
3. **Aufstellen der Zustandsübergangsfunktion**
   - Wertetabelle
4. **Aufstellen der Ausgabefunktion**
   - Wertetabelle

Die Schritte 2-4 können automatisch erfolgen!

Automat (Moore)



E: Eingänge

A: Ausgänge

Z: Zustandsübergangsfunktion
S: Speicher (Flipflops)
A: Ausgangsfunktion

Typen

- **Neue Typen können nach der Definition**
  - wie schon bekannte Typen verwendet werden
  - lokal definiert werden, z.B. in einer *architecture*
  - global in einem *package* definiert werden

- **Möglichkeiten der Definition**
  - Aufzählung einzelner Elemente (bekannt: *std_logic*)
  - Einschränkung eines Typs (bekannt: *positive* aus *integer*)
  - Organisation als Feld (bekannt: *bit_vector* aus *bit*)
  - Zusammenstellung zu einer Einheit

Aufzählung

Der Typ ist eine Menge einzeln benannter Elemente

- Die tatsächliche Zuordnung der Elemente zu einer digitalen Repräsentation muss nicht angegeben werden, ist aber möglich
- Die Elemente haben eine Ordnung (Vorgänger/Nachfolger)
- Aufzählungselemente sind für jeden Typ unabhängig deklariert (d.h. derselbe Elementname kann in unterschiedlichen Typen verwendet werden und bedeutet dann jeweils etwas anderes)

```
type name is (element_1, element_2, …, element_n);

architecture mfm of ampel is
   type ampelfarbe is (aus, rot, rotgelb, gruen, gelb);
   signal zustand: ampelfarbe;

begin
   rote_led <= '1' when (zustand = rot or zustand = rotgelb) else '0';
```

Attribute 1

- **Attribute sind Eigenschaften von Objekten**
  - In VHDL ist (fast) alles ein Objekt, aber häufigste Nutzung bei Feldern
  - Standardattribute sind im Sprachumfang bzw. in Standardpackages vordefiniert
  - „user defined attributes“ können beliebig selbst definiert und benutzt werden. Das wird aber nur selten genutzt
  - Viele Hersteller nutzen „user defined attributes“ zur Steuerung der Werkzeuge. Diese Attribute sind aber dann nicht portabel.
    Beispiel: Angabe der Anschlussnummer am FPGA als Eigenschaft eines Signals in der Portliste der entity
  - Die Werte aller Attribute müssen zum Zeitpunkt der Kompilation bekannt sein (wie bei den generics)

Attribute 2

- Definition eines „user defined attribute“ durch Angabe des Namens des Attributs und des Typs des Attributs.
- Anhängen eines solchen Attributs an ein Objekt mittels einmaliger Zuweisung. Dabei müssen Name und Typ des Objekts angegeben werden.
- Die Zuweisung erfolgt im deklarativen Teil der Einheit, also bei einer *architecture* vor dem *begin*
  Auch in einer *entity* sind Deklarationen möglich. Sie folgen dann der *port-* bzw. *generic-Liste.*

```
-- Definition eines “user defined attribute“, hier als String
attribute pin: string;
-- Zuweisung des Attributs mit Wert an ein Signal mit dem Namen clk
attribute pin of clk: signal is "H16";
```

Feld

- **Deklaration durch neuen Typ**
  - Anzahl der Elemente kann offen bleiben (unconstrained)

```
type name is array (range x to y) of elementyp;
type name is array (range x downto y) of elementyp;
type name is array (integertyp range <>) of elementtyp;

type byte is array (7 downto 0) of std_logic;     -- Ein Byte hat immer 8 Bits
type memory is array (natural range <>) of byte; -- Speichergröße noch unbestimmt

signal ledmatrix: memory (0 to 15);               -- Speicher für 16 Bytes

ledmatrix(11) <= "00101010";                -- Zugriff auf Byte 11
ledmatrix(12)(2 downto 0) <= "001";         -- Zugriff auf 3 Bits in Byte 12
```

- **Nutzung wie bei Vektoren (das sind nur vordefinierte arrays)**
  - Beliebige Bereiche (range) können in einer Anweisung bearbeitet werden
  - Zweidimensionale Felder werden bei der Synthese meist Speicher

Attribute 3

- Zugriff auf ein Attribut durch Anhängen des Namens des Attributs an den Namen des Objekt mittels Apostroph
- Auswahl einiger Attribute für Array-Objekte (A):

```
A'LEFT     the leftmost subscript of array A or constrained array type.
A'RIGHT    the rightmost subscript of array A or constrained array type.
A'HIGH     the highest subscript of array A or constrained array type.
A'LOW      the lowest subscript of array A or constrained array type.
A'RANGE    the range  A'LEFT to A'RIGHT  or  A'LEFT downto A'RIGHT .
A'LENGTH   the integer value of the number of elements in array A.
```

```
entity demo is
  Port (
   ausgabe: out std_logic_vector(1 to 3); -- Ausgabesignal

Architecture mfm of demo is
  -- Deklaration eines internen Signals mit gleichem Bereich
  signal aus_intern: std_logic_vector(ausgabe'range);
```

Struktur

- **Deklaration durch neuen Typ**

```
type record_typname is
record
  element1_name : typ1;
  element2_name : typ2;
end record;
```

- **Nutzung**

```
signal signame: record_typname;
…
signame.element1_name  <= …
```

- **Nutzung in der component/entity**

  Bis einschließlich VHDL2008 hat ein *record* insgesamt eine Richtung.
  Ab VHDL2019 können Elemente unterschiedliche Richtungen haben.

Funktion

Funktionen werden zwar wie in einer Programmier-sprache benutzt, werden aber bei der Schaltungs-synthese bei jedem Aufruf instanziiert.

N Funktionsaufrufe -> N-facher Aufwand

Deklaration / Definition meist in einem package, aber auch lokal in einer architecture möglich

In der Funktion ist die sequentielle Umgebung aktiv

Funktion - Syntax

- Aufbau ähnlich einer Kombination aus entity und architecture
- Die Signalliste entspricht der Portliste einer entity,
  aber nur die Richtung in ist zulässig
- Hat immer einen Rückgabewert
- Kann lokale Deklarationen enthalten

```
function name (signalliste) return datentyp is
  -- lokale Deklarationen
begin
  --Funktionskörper
  return wert; -- Rückgabe ist zwingend
end;
```

Schleifen

- Schleifen werden **bei der Kompilation** ausgeführt
- Jeder Schleifendurchlauf erzeugt (in der Regel) Hardware, d.h. **Schaltungsaufwand**
- Diese Hardware arbeitet dann **nebenläufig**
- Die Schleifenvariable muss nicht deklariert werden und ist **außerhalb der Schleife nicht sichtbar**
- **Syntaktische Unterscheidung** zwischen Anwendung in der sequentiellen und nebenläufigen Umgebung
- Abfrage der Variable mit **if** für Sonderfälle ist **in beiden Fällen** möglich

Schleifen

- In der sequentiellen Umgebung: for loop

```
for i in range
loop
  sequentielle Anweisungen;
  exit; -- when bedingung
  next; -- when bedingung;
end loop;
```

- Range (z.B. 0 to 7) muss konstant sein
- exit und next können optional eine Bedingung haben
- Schleifen können geschachtelt werden

- In der nebenläufigen Umgebung: for generate loop

```
label: for i in range
generate
  multireg : reg port map (clk, d(i), q(i));
end generate label;
```