

Digitaltechnik

Wintersemester 2025/2026

Vorlesung 9



TECHNISCHE
UNIVERSITÄT
DARMSTADT












ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

- 1 SystemVerilog für sequentielle Logik
- 2 Zeitverhalten synchroner sequentieller Logik
- 3 Parallelität



Harris 2016
Kap. 3.5, 4.4,
3.6










Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

Abgabefrist für Hausaufgabe D zu
Vorlesungen 07 und 08 nächste Woche
Freitag 23:59!
Wöchentliches Moodle-Quiz nicht vergessen!

1 SystemVerilog für sequentielle Logik

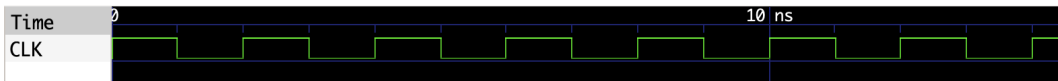
2 Zeitverhalten synchroner sequentieller Logik

3 Parallelität

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

Das Clock-Signal

Beispiel für Verzögerungen und sequentielle Logik



clock.sv

```
1 `timescale 1 ns / 10 ps
2
3 module clock(output logic CLK);
4
5     always begin
6         CLK = 1;
7         #1;
8         CLK = 0;
9         #1;
10    end
11
12 endmodule
```

Code
synthetisiert
nicht!

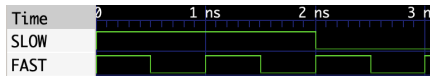
- ``timescale <base> / <precision>`
vor Modul spezifiziert
- ⇒ Verzögerung # `<tval>`; für
`<tval>·<base>`
- Genauigkeit `<precision>`, auf welche
die Verzögerungszeit gerundet wird
- Block `<always>` heißt auch Prozess
- ⇒ Wird endlos wiederholend ausgeführt

Synthese: Eigener (externer) Baustein für Clock-Signal; Code hier simuliert nur Clock

- always führt den nachfolgenden Block in einer Endlosschleife aus
- alle always Blöcke werden parallel (nebenläufig) ausgeführt

twoClocks.sv

```
1 `timescale 1 ns / 10 ps
2
3 module twoClocks(output logic SLOW, FAST);
4
5     always begin
6         SLOW = 1;
7         #2;
8         SLOW = 0;
9         #2;
10    end
11
12    always begin
13        FAST = 1;
14        #0.5;
15        FAST = 0;
16        #0.5;
17    end
18
19 endmodule
```



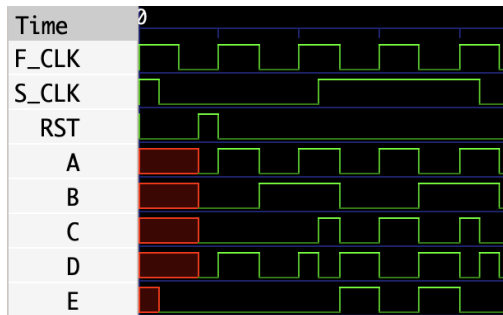


Häufig wollen wir etwas nur berechnen, wenn ein bestimmtes Ereignis auftritt, z.B. eine steigende Taktflanke. Dazu gibt es folgende Zusätze für `always` Blöcke:

- `@ <expr>` wartet auf Änderung des kombinatorischen Ausdrucks `<expr>`
- `@(posedge <expr>)` wartet auf steigende Flanke von `<expr>`
($0 \rightarrow 1, x \rightarrow 1, z \rightarrow 1, 0 \rightarrow z, 0 \rightarrow x$)
- `@(negedge <expr>)` wartet auf fallende Flanke von `<expr>`
($1 \rightarrow 0, x \rightarrow 0, z \rightarrow 0, 1 \rightarrow z, 1 \rightarrow x$)
- `@(<event1>, <event2>)` wartet auf Eintreten eines der aufgelisteten Ereignisse
 - wird auch als *Sensitivitätsliste* bezeichnet
- `@*` wartet auf Änderung eines der im `always` Block gelesen Signale

events.sv

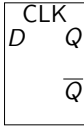
```
1 module events(input logic F_CLK, S_CLK,  
2               RST,  
3               output logic A, B, C, D, E);  
4  
5   always @(posedge RST) begin  
6     A = 0;  
7     B = 0;  
8     C = 0;  
9     D = 0;  
10    E = 0;  
11  end  
12  
13  // Kurzschreibweise wenn für einzelnes  
14  // Kommando:  
15  always @F_CLK      A = ~A;  
16  always @(negedge F_CLK) B = ~B;  
17  always @(F_CLK & S_CLK) C = ~C;  
18  always @(F_CLK, S_CLK) D = ~D;  
19  // Würde beliebig schnell oszillieren:  
20  //always @*          E = ~E;  
21  always @*          E = D & S_CLK;  
22  
23 endmodule
```





- Bisher: **Blockierende Zuweisungen** $\langle \text{signal} \rangle = \langle \text{expr} \rangle;$
 - $\langle \text{expr} \rangle$ wird ausgewertet und an $\langle \text{signal} \rangle$ zugewiesen, *bevor* nächste Zuweisung behandelt wird
 - ⇒ blockierende Zuweisungen werden in gegebener Reihenfolge (sequentiell) abgehandelt
- Neu: **Nicht-blockierende Zuweisungen** $\langle \text{signal} \rangle \leq \langle \text{expr} \rangle;$
 - $\langle \text{expr} \rangle$ aller nicht-blockierenden Zuweisungen in einer Sequenz werden ausgewertet, aber *noch nicht* an $\langle \text{signal} \rangle$ zugewiesen, sondern nur vorgemerkt
 - ⇒ nicht-blockierende Zuweisungen werden nebenläufig (parallel) abgehandelt
- Beispiele dazu später!

D-Latches in SystemVerilog



latch.sv

```
1 module latch (input logic CLK,D,  
2               output logic Q);  
3  
4     always @* if (CLK) Q <= D;  
5  
6 endmodule
```

latch_better.sv

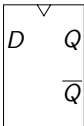
```
1 module latch (input logic CLK,D,  
2               output logic Q);  
3  
4     always_latch if (CLK) Q <= D;  
5  
6 endmodule
```



D-Flip-Flops in SystemVerilog



ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

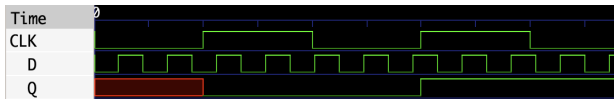
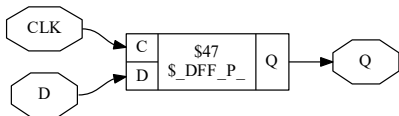


dff.sv

```
1 module dff (input logic CLK,D,  
2             output logic Q);  
3  
4     always @(posedge CLK) Q <= D;  
5  
6 endmodule
```

dff_better.sv

```
1 module dff (input logic CLK,D,  
2             output logic Q);  
3  
4     always_ff @(posedge CLK) Q <= D;  
5  
6 endmodule
```



Weitere Beispiele

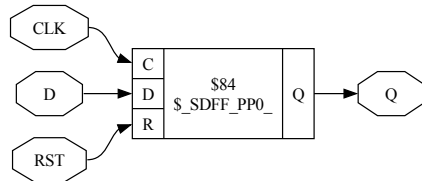
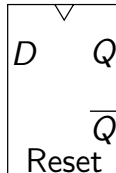
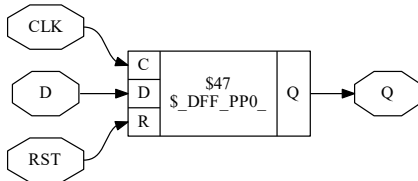
Rücksetzbare D-Flip-Flops

dffar.sv

```
1 // asynchron rücksetzbar
2 module dffar (input logic CLK,RST,D,
3               output logic Q);
4
5     always_ff @(posedge CLK, posedge RST) begin
6         if (RST) begin
7             Q <= 0;
8         end else begin
9             Q <= D;
10        end
11    end
12
13 endmodule
```

dffr.sv

```
1 // synchron rücksetzbar
2 module dffr (input logic CLK,RST,D,
3               output logic Q);
4
5     always_ff @(posedge CLK) begin
6         if (RST) begin
7             Q <= 0;
8         end else begin
9             Q <= D;
10        end
11    end
12
13 endmodule
```



Weitere Beispiele

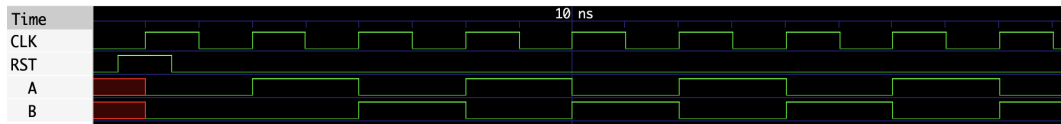
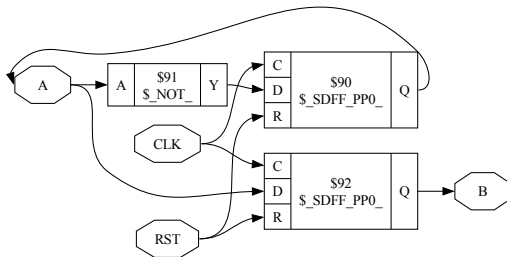
Schaltung mit D-Flip-Flops



ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

example.sv

```
1 module example(input logic CLK, RST,  
2               output logic A, B);  
3  
4   always_ff @(posedge CLK) begin  
5     if (RST) begin  
6       A <= 0;  
7       B <= 0;  
8     end else begin  
9       A <= ~A;  
10      B <= A;  
11    end  
12  end  
13  
14 endmodule
```



- `always`
 - Prozess, wird endlos oder durch Ereignis (z.B. `@(CLK)`) getriggert ausgeführt
 - `always_comb`
 - Funktioniert (ähnlich) wie `always @*` (ausgeführt, wenn RHS Variable sich ändert)
 - **Für kombinatorische Logik (if/else, ...) verwenden!** (oder auch `assign`)
 - `always_latch`
 - Funktioniert (ähnlich) wie `always @*`
 - Wird für synchrone Schaltungen kaum benutzt
 - `always_ff @(<event>)`
 - Funktioniert (ähnlich) wie `always @(<event>)`
 - **Für sequentielle Logik mit Flip-Flops verwenden!**
- ⇒ Spezialisierte Blöcke nutzen, damit Synthese-Tools die Absicht des Designs besser verstehen und auf Fehler hinweisen können
- In alten Versionen von Icarus-Verilog werden spezialisierte Blöcke manchmal nicht unterstützt, dann darf/muss man `always` verwenden

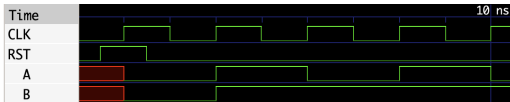
Blockierende vs nicht-blockierende Zuweisungen



ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

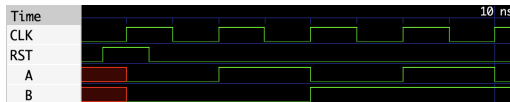
sequential.sv

```
1 module sequential(input logic CLK, RST,  
2                   output logic A, B);  
3  
4     always_ff @(posedge CLK) begin  
5         if (RST) begin  
6             A = 0;  
7             B = 0;  
8         end else begin  
9             A = ~A;  
10            B = A | B;  
11        end  
12    end  
13  
14 endmodule
```



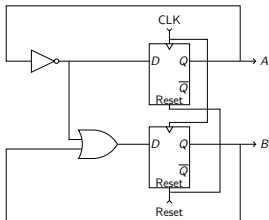
parallel.sv

```
1 module parallel(input logic CLK, RST,  
2                 output logic A, B);  
3  
4     always_ff @(posedge CLK) begin  
5         if (RST) begin  
6             A <= 0;  
7             B <= 0;  
8         end else begin  
9             A <= ~A;  
10            B <= A | B;  
11        end  
12    end  
13  
14 endmodule
```



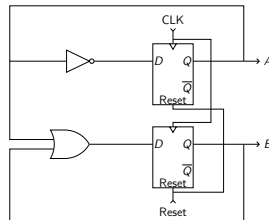
sequential.sv

```
1 module sequential(input logic CLK, RST,  
2                   output logic A, B);  
3  
4     always_ff @(posedge CLK) begin  
5       if (RST) begin  
6         A = 0;  
7         B = 0;  
8       end else begin  
9         A = ~A;  
10        B = A | B;  
11      end  
12    end  
13  
14 endmodule
```



parallel.sv

```
1 module parallel(input logic CLK, RST,  
2                 output logic A, B);  
3  
4     always_ff @(posedge CLK) begin  
5       if (RST) begin  
6         A <= 0;  
7         B <= 0;  
8       end else begin  
9         A <= ~A;  
10        B <= A | B;  
11      end  
12    end  
13  
14 endmodule
```





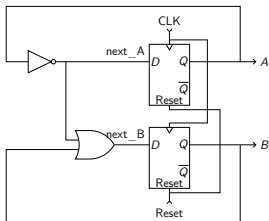
- interne Zustände
 - innerhalb von `always_ff @(posedge CLK)`
 - mit nicht-blockierenden Zuweisungen (`<=`)
 - *möglichst* nur ein/wenige Zustände pro `always_ff` block
- einfache kombinatorische Logik durch nebenläufige Zuweisungen (`assign`)
- komplexere kombinatorische Logik
 - innerhalb von `always_comb`
 - mit blockierenden Zuweisungen (`=`)
- ein Signal darf **nicht**
 - von mehreren nebenläufigen Prozessen (`assign` oder `always`) beschrieben werden
 - innerhalb eines `always` Blocks mit blockierenden und nicht-blockierenden Zuweisungen beschrieben werden

Allgemeine Regeln für Signalzuweisungen

Beispiel

sequential.sv

```
1 module sequential(input logic CLK, RST,  
2                   output logic A, B);  
3  
4     always_ff @(posedge CLK) begin  
5         if (RST) begin  
6             A = 0;  
7             B = 0;  
8         end else begin  
9             A = ~A;  
10            B = A | B;  
11        end  
12    end  
13  
14 endmodule
```












sequential_better.sv

```
1 module sequential(input logic CLK, RST,  
2                   output logic A, B);  
3  
4     logic next_A, next_B; // inputs to DFF  
5  
6     always_ff @(posedge CLK) begin  
7         if (RST) begin  
8             A <= 0; // replace = by <=  
9             B <= 0;  
10        end else begin  
11            A <= next_A; // overwrite previous state  
12            B <= next_B;  
13        end  
14    end  
15  
16    assign next_A = ~A;  
17    assign next_B = next_A | B;  
18  
19 endmodule
```

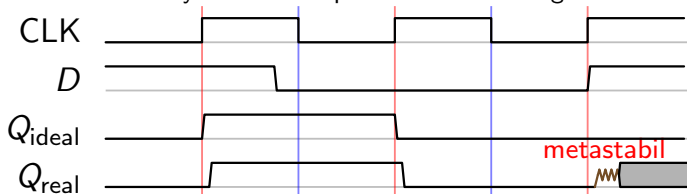
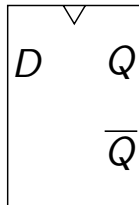
1 SystemVerilog für sequentielle Logik

2 Zeitverhalten synchroner sequentieller Logik

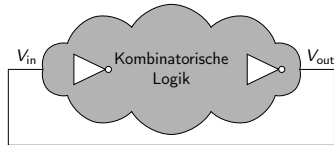
3 Parallelität

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen

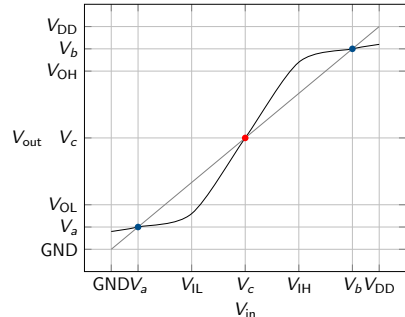
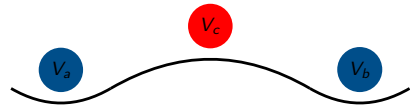
- Flip-Flop übernimmt D zur steigenden Taktflanke
- Was passiert bei zeitgleicher Änderung von D und CLK?
- bisher vereinfachte Annahme:
 - Wert unmittelbar vor der Taktflanke wird übernommen
- Aber:
 - Was heißt „unmittelbar“?
 - Wie schnell wird neuer Zustand am Ausgang sichtbar?
 - Was muss daher bei synchronen sequentiellen Schaltungen beachtet werden?



- In Digitaltechnik:
 - zeitlich begrenzter und undefinierter Zustand
 - geht nach zufälliger Verzögerung in einen stabilen Zustand über
- Beispiel:



- Rückkopplung stabil für $V_{out} = V_{in}$
 - V_a repräsentiert 0
 - V_b repräsentiert 1
 - V_c im „verbotenen“ Spannungsbereich
 - kleine Änderung an V_{in}
→ große Änderung an V_{out}



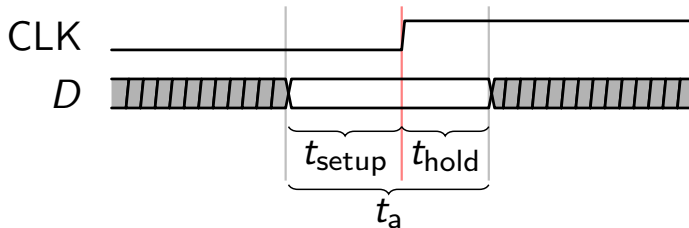
- Dateneingang D muss im Abtast-Zeitfenster um Taktflanke stabil sein, um Metastabilität zu vermeiden

t_{setup} Zeitintervall vor Taktflanke, in dem D stabil sein muss ("setup time")

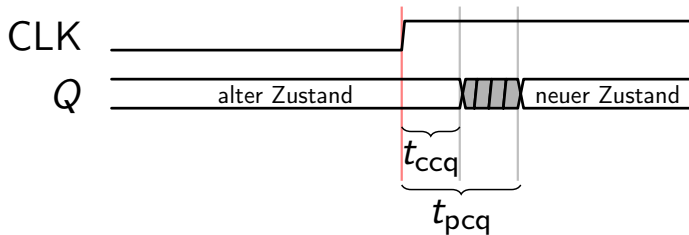
t_{hold} Zeitintervall nach Taktflanke, in dem D stabil sein muss ("hold time")

t_a Abtastzeitfenster: $t_a = t_{\text{setup}} + t_{\text{hold}}$ ("aperture time")

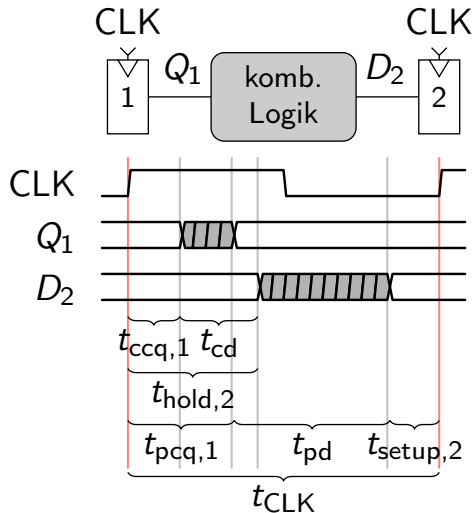
- Größenordnung: 10 ps



- Verzögerung des Registerausgangs relativ zur steigenden Taktflanke
 - Kontaminationsverzögerung (t_{ccq}): kürzeste Zeit bis Q umschaltet ("contamination delay clock-to-Q")
 - Laufzeitverzögerung (t_{pcq}): längste Zeit bis Q sich stabilisiert ("propagation delay clock-to-Q")
- Größenordnung: 10 ps



- kombinatorische Logik zwischen zwei Registern hat min. Verzögerung t_{cd} und max. Verzögerung t_{pd}
 - D_2 abhängig von Verzögerungen der Gatter und des *ersten* Registers
- ⇒ Timing-Bedingungen des *zweiten* Registers müssen erfüllt werden
- $t_{ccq,1} + t_{cd} \geq t_{hold,2}$
 - $t_{pcq,1} + t_{pd} + t_{setup,2} \leq t_{CLK}$
- ⇒ maximale Taktrate wird durch *kritischen Pfad* bestimmt
- $f_{CLK} = \frac{1}{t_{CLK}} \leq \frac{1}{t_{pcq} + t_{pd} + t_{setup}}$




■ Timing-Vorgaben:

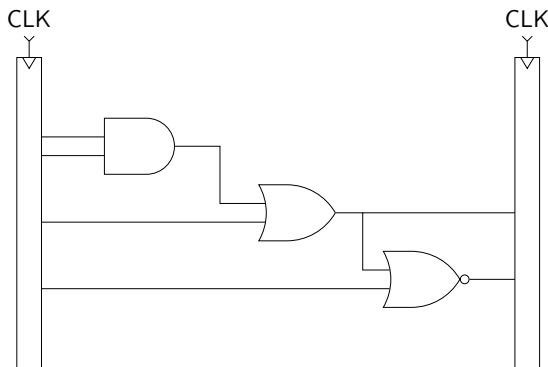
- $t_{ccq} = 30 \text{ ps}$
- $t_{pcq} = 50 \text{ ps}$
- $t_{setup} = 60 \text{ ps}$
- $t_{hold} = 70 \text{ ps}$
- $t_{cd,Gatter} = 25 \text{ ps}$
- $t_{pd,Gatter} = 35 \text{ ps}$

■ kombinatorischer Pfad:

- $t_{cd} = 25 \text{ ps}$
- $t_{pd} = 3 \cdot 35 \text{ ps} = 105 \text{ ps}$

■ Timing-Bedingungen:

- $f_{CLK} \leq \frac{1}{t_{pcq} + t_{pd} + t_{setup}} = \frac{1}{215 \text{ ps}} = 4,65 \text{ GHz}$
- $t_{ccq} + t_{cd} = 55 \text{ ps} < t_{hold}$ 



Beispiel: Beheben der verletzten Hold-Zeit Anforderung



■ Timing-Vorgaben:

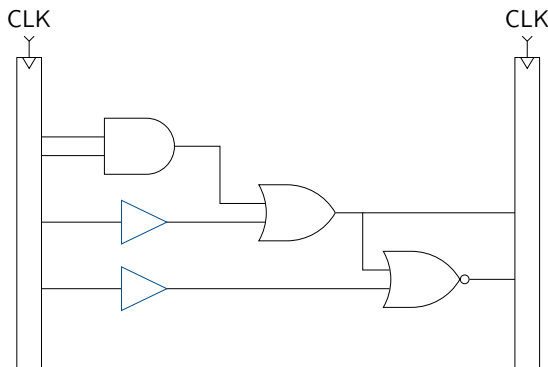
- $t_{ccq} = 30 \text{ ps}$
- $t_{pcq} = 50 \text{ ps}$
- $t_{setup} = 60 \text{ ps}$
- $t_{hold} = 70 \text{ ps}$
- $t_{cd,Gatter} = 25 \text{ ps}$
- $t_{pd,Gatter} = 35 \text{ ps}$

■ kombinatorischer Pfad:

- $t_{cd} = 50 \text{ ps}$
- $t_{pd} = 3 \cdot 35 \text{ ps} = 105 \text{ ps}$

■ Timing-Bedingungen:

- $f_{CLK} \leq \frac{1}{t_{pcq} + t_{pd} + t_{setup}} = \frac{1}{215 \text{ ps}} = 4,65 \text{ GHz}$
- $t_{ccq} + t_{cd} = 80 \text{ ps} > t_{hold} \checkmark$



- asynchrone Eingänge:

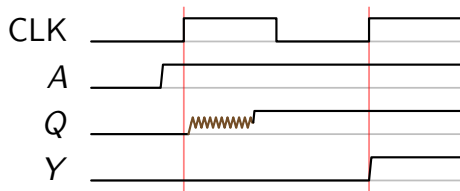
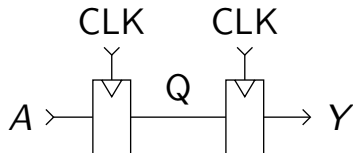
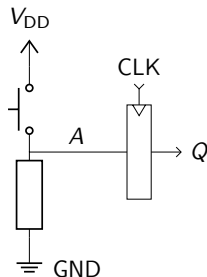
- Eingaben
- Kommunikationssignale von externen ICs

⇒ Timing-Bedingungen können nicht garantiert werden

- Schieberegister für Synchronisation

- erstes Flip-Flop kann metastabil werden
- kippt i.d.R. vor nächster Taktflanke in stabilen Zustand










⇒ zweites Flip-Flop wird nicht metastabil



1 SystemVerilog für sequentielle Logik

2 Zeitverhalten synchroner sequentieller Logik

3 Parallelität

Anwendungs- software		Programme
Betriebs- systeme		Gerätetreiber
Architektur		Befehle Register
Mikro- architektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital- schaltungen		UND Gatter Inverter
Analog- schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen



- räumliche Parallelität
 - mehrere Aufgaben durch vervielfachte Hardware gleichzeitig bearbeiten
 - zeitliche Parallelität
 - Aufgabe in mehrere Unteraufgaben aufteilen
 - Unteraufgaben parallel ausführen
 - Beispiel: Fließbandprinzip bei Autofertigung („Pipelining“)
 - nur eine Station pro Arbeitsschritt
 - alle unterschiedlichen Arbeitsschritte für mehrere Autos parallel ausgeführt
- ⇒ zeitliche Parallelität

Datensatz: Vektor aus Eingabewerten, zu denen ein Vektor aus Ausgabewerten berechnet wird

Latenz: Zeit von der Eingabe eines Datensatzes bis zur Ausgabe des zugehörigen Ergebnisses

Durchsatz: Anzahl von Datensätzen, die pro Zeiteinheit bearbeitet werden können
⇒ Parallelität erhöht Durchsatz

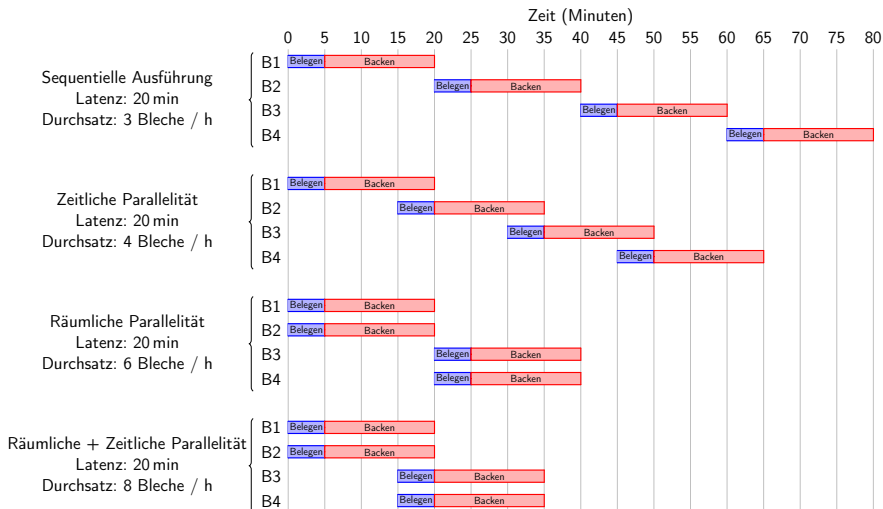
- Annahmen:
 - genug Teig ist fertig
 - 5 Minuten zum Belegen eines Bleches
 - 15 Minuten Backzeit
- *sequentiell*: ein Blech nach dem anderen belegen und backen
- *zeitlich parallel*: nächstes Blech belegen, während erstes noch im Ofen ist
- *räumlich parallel*: zwei Bäcker:innen, jeweils mit eigenem Ofen
- räumliche und zeitliche Parallelität kombinierbar



Beispiel: Plätzchen backen

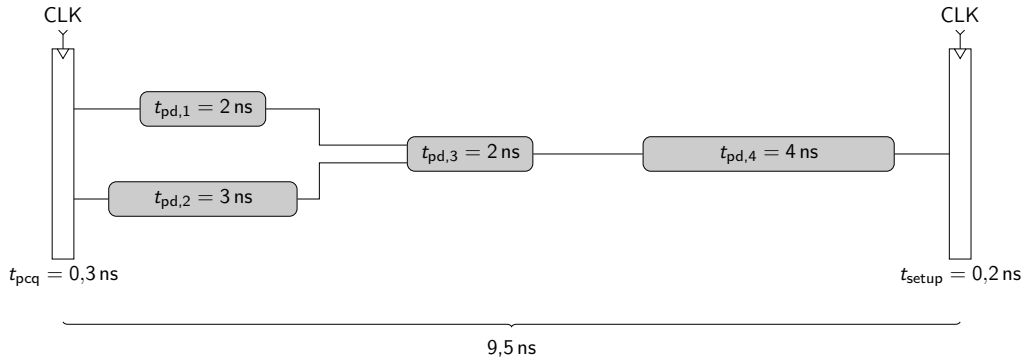


ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING



Beispiel: Pipelining in Schaltungen

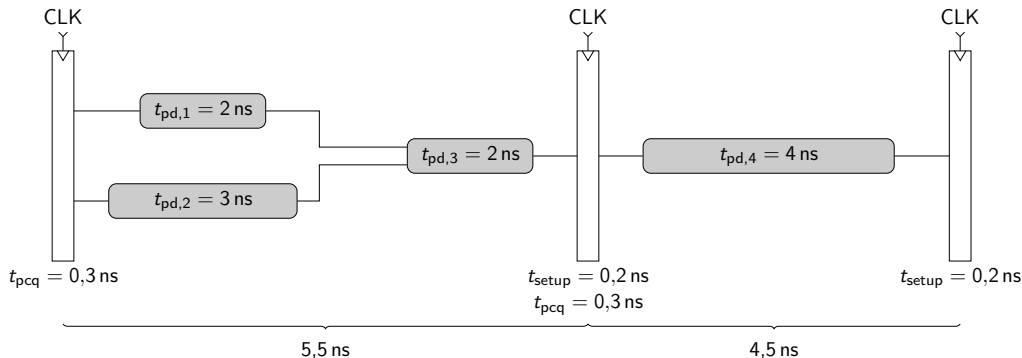
Ohne Pipeline-Register



- $f_{CLK} \leq \frac{1}{0,3+3+2+4+0,2\text{ ns}} = \frac{1}{9,5\text{ ns}} = 105\text{ MHz}$
- Latenz: 1 Takt = $9,5\text{ ns}$

Beispiel: Pipelining in Schaltungen

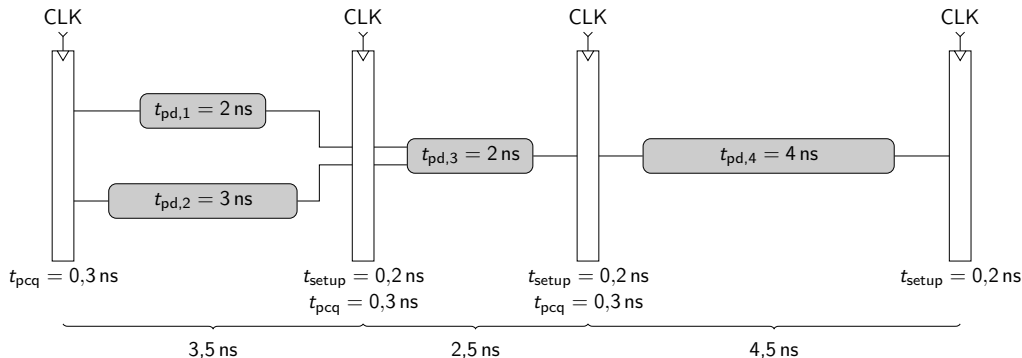
Zwei Pipeline-Stufen



- $f_{CLK} \leq \min(\frac{1}{5,5\text{ ns}}, \frac{1}{4,5\text{ ns}}) = \frac{1}{5,5\text{ ns}} = 182\text{ MHz}$
- Latenz: 2 Takte = $2 \cdot 5,5\text{ ns} = 11\text{ ns}$

Beispiel: Pipelining in Schaltungen

Drei Pipeline-Stufen



- $f_{CLK} \leq \min\left(\frac{1}{3,5 \text{ ns}}, \frac{1}{2,5 \text{ ns}}, \frac{1}{4,5 \text{ ns}}\right) = \frac{1}{4,5 \text{ ns}} = 222 \text{ MHz}$
- Latenz: 3 Takte = $3 \cdot 4,5 \text{ ns} = 13,5 \text{ ns}$












- Pipelinestufen sollten möglichst gleich lang sein („ausbalanciert“)
 - längste Stufe bestimmt maximale Taktfrequenz f_{CLK}
 - Latenz = # Pipelinestufen / Taktfrequenz
- mehr Pipelinestufen
 - höherer Durchsatz (mehr Ergebnisse pro Zeiteinheit), da höhere Taktfrequenz
 - aber auch höhere Latenz (länger auf Ergebnis warten)
 - ⇒ lohnt sich nur, wenn viele Datensätze bearbeitet werden müssen
- Probleme bei Abhängigkeiten zwischen Teilaufgaben
 - bspw. erst Backergebnis prüfen, bevor nächstes Blech belegt wird
- Ausführliche Behandlung s. Befehlsverarbeitung von Prozessoren in LV Rechnerorganisation

- 1 SystemVerilog für sequentielle Logik
- 2 Zeitverhalten synchroner sequentieller Logik
- 3 Parallelität

nächste Vorlesung beinhaltet

- Endliche Zustandsautomaten (FSM)
 - Konzept, Notationen
 - Moore vs. Mealy Automaten
- SystemVerilog für Zustandsautomaten

Hausaufgabe D zu Vorlesungen 07 und 08 muss bis nächste Woche Freitag 23:59 abgegeben werden.
Wöchentliches Moodle-Quiz nicht vergessen!

Anwendungssoftware		Programme
Betriebssysteme		Gerätetreiber
Architektur		Befehle Register
Mikroarchitektur		Datenpfade Steuerung
Logik		Addierer Speicher
Digital-schaltungen		UND Gatter Inverter
Analog-schaltungen		Verstärker Filter
Bauteile		Transistoren Dioden
Physik		Elektronen