Literatur

1. David Money Harris & Sarah L. Harris: "Digital Design and Computer Architecture", Morgan Kaufmann 2007

2. John F. Wakerly: "Digital Design, Principles & Practices", Prentice-Hall 2001

3. David A. Patterson, John L. Hennessy: "Computer Organization & Design, The Hardware/Software Interface", Morgan Kaufmann 1998 (gibt's auch als PDF im Netz)

4. Peter J. Ashenden: "The Designer's Guide to VHDL", Morgan Kaufmann 1996

5. Donald E. Thomas, Philip R. Moorby: "The Verilog Hardware Description Language", Kluwer Academic Publishers 2002

Übersicht

1. Hardware-Entwurf
   - Kombinatorische Digitalschaltungen ("Schaltnetze")
   - Sequentielle Digitalschaltungen ("Schaltwerke")
   - Moore- und Mealy-Automaten
   - Speicher und programmierbare Logikbausteine
2. Rechnerarchitektur
   - Performance-Messung
   - Instruction-Set-Architecture

Arithmetik

Datenpfad und Steuerung

Pipelining

Caches

~~Virtueller Speicher~~

Eingabe / Ausgabe

~~3. Eingebettete Systeme und ihre Umgebung~~

~~Meßprinzipien~~

~~Sensoren~~

~~Wandler (ADC / DAC)~~

~~Aktoren~~

~~Kommunikation~~

4. Werkzeuge (werden im Praktikum behandelt)

Hardware-Beschreibungssprachen

Simulatoren

Synthesewerkzeuge

---

3 Testfragen:

1) Was ist der Unterschied zwischen einem Schaltnetz und einem Schaltwerk?

2) Was zeichnet eine synchrone Schaltung aus?

3) Was ist "Metastabilität"? Muß man darauf überhaupt achten?

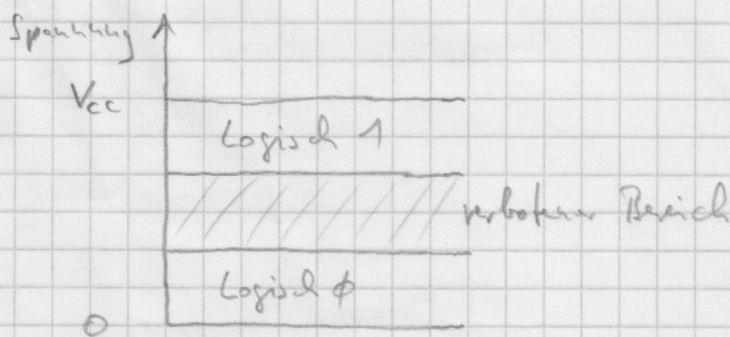
1. Hardware-Entwurf

1.1. Analog vs. Digital

Analogrechner: Die Rechengrößen werden durch eine physikal. Größe (z.B. Spannung) repräsentiert, die jeden Wert zwischen zwei Grenzen (z.B. ± 10V) annehmen kann und die der Rechengröße bis auf eine Proportionalitätskonstante gleich ist.

Digitalrechner: Die Rechengrößen werden durch eine gewisse Zahl (z.B. 32) von Elementen mit diskreten Zuständen (meistens zwei) dargestellt. Die zwei Zustände heißen FALSE und TRUE (oder kürzer: $\emptyset$ und 1).

Heißt das, wir finden Nullen und Einsen in unseren Digitalschaltungen? Nein! Zwei Spannungsbereiche dienen zur Darstellung der $\emptyset$ und der 1:

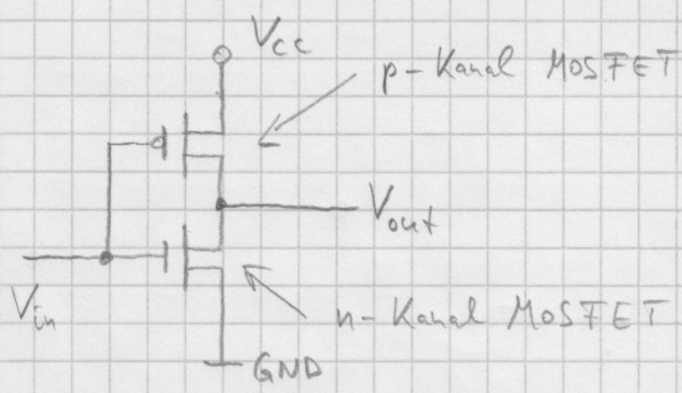


1.2. Gatter

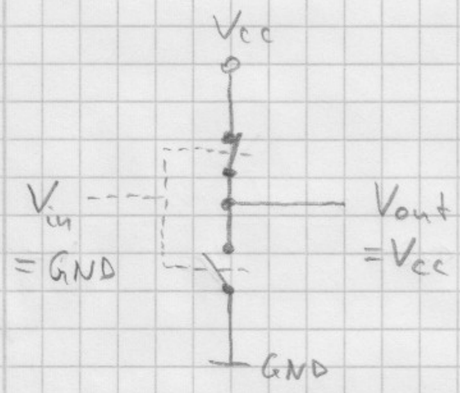
3 Gattertypen reichen aus, um jede Boole'sche Funktion

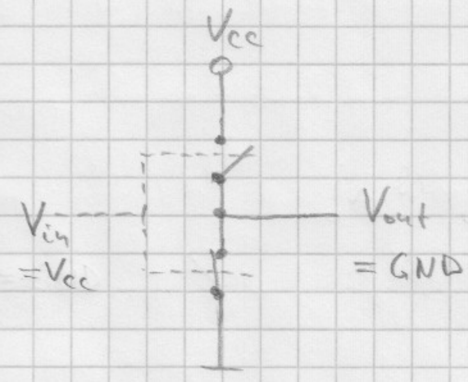
$(x_1, x_2, ..., x_n) \mapsto f(x_1, x_2, ..., x_n)$ zu realisieren.

a) NICHT-Gatter ("Inverter")



Schaltung

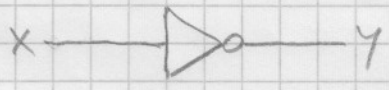




Wirkungsweise

x	y
0	1
1	0

Wahrheitstabelle



Schaltsymbol

Algebraische Schreibweise: $y = \overline{x}$

$y = x'$

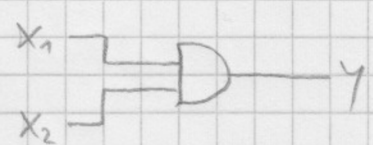
Verilog:

a) `not g1(y, x);` ← "Strukturbeschreibung"

b) `assign y = ~x;` ← "Verhaltensbeschreibung"

b) UND-Gatter

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

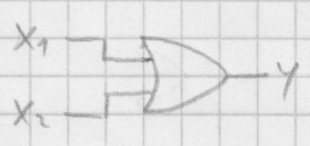


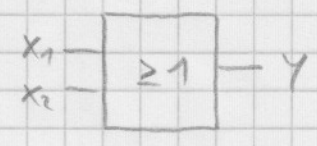
$y = x_1 \cdot x_2$

```
and g1(y, x1, x2);
assign y = x1 & x2;
```

c) ODER - Gatter

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



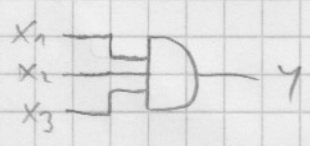


$y = x_1 + x_2$

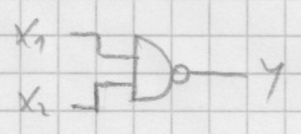
```
or g1(y, x1, x2);
assign y = x1 | x2;
```

Bem.:

a) Mehr als zwei Eingänge sind möglich; z.B. $y = x_1 \cdot x_2 \cdot x_3$

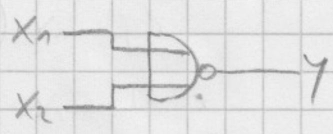


b) NAND ≡ "not and" : $y = \overline{x_1 \cdot x_2}$



x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

NOR ≡ "not or" : $y = \overline{x_1 + x_2}$

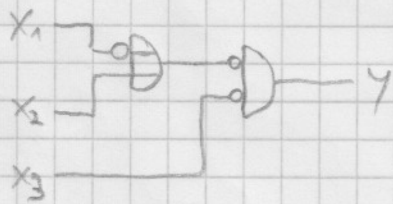


x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	0

XOR ≡ "exclusive or" : $y = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

c) Der "Inverterkringel" kann auch an die Eingänge von Gattern geschrieben werden; z.B.:



$$y = \overline{(\overline{x_1} + x_2)} \cdot \overline{x_3}$$

d) Die analoge Wirklichkeit schlägt zurück:

– Jeder Eingang belastet den treibenden Ausgang mit einem kleinen Strom ⇒ Man kann nicht bel. viele Eingänge an einen Ausgang hängen ("Fan-out" ≈ 20)

– Ein Signalwechsel am Eingang eines Gatters überträgt sich nur mit einer gewissen Verzögerung an seinen Ausgang ("Propagation Delay" ≈ nsec)

1.3. Kombinatorische vs. sequentielle Logik

Kombinatorische Logik ("Schaltnetze"):

- Werte der Ausgänge hängen nur von den momentanen Werten der Eingänge ab, hat also kein "Gedächtnis"
- Bel. viele Gatter und Signalverzweigungen, aber <u>keine Rückkopplungsschleifen</u> (= Pfade, die vom Ausgang eines Gatters direkt oder indirekt über andere Gatter wieder zu einem Eingang dieses Gatters zurückführen)

Sequentielle Logik ("Schaltwerke"):

- Werte der Ausgänge hängen außer von den momentanen Werten der Eingänge auch von bel. vielen vorhergehenden Werten ab, d.h. die Schaltung ist "zustandsbehaftet" (hat also "Gedächtnis")
- Dieses Verhalten wird durch "Rückkopplung" von Ausgängen einer Schaltung auf Eingänge derselben Schaltung erreicht. Genaueres später!

1.4. Kombinatorische Logik

1.4.1. Boole'sche Algebra (Schaltalgebra)

$$x + 0 = x \qquad x \cdot 1 = x$$

$$x + 1 = 1 \qquad x \cdot 0 = 0$$

$$x + x = x \qquad x \cdot x = x$$

$$\overline{\overline{x}} = x$$

$$x + \overline{x} = 1 \qquad x \cdot \overline{x} = 0$$

$$x + y = y + x \qquad x \cdot y = y \cdot x$$

$$(x + y) + z = x + (y + z) \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$x \cdot (y + z) = x \cdot y + x \cdot z \qquad x + (y \cdot z) = (x + y) \cdot (x + z)$$

$$x + x \cdot y = x \qquad x \cdot (x + y) = x$$

$$x \cdot y + x \cdot \overline{y} = x \qquad (x + y) \cdot (x + \overline{y}) = x$$

$$\overline{x + y} = \overline{x} \cdot \overline{y} \qquad \overline{x \cdot y} = \overline{x} + \overline{y}$$

1.4.2. Normalformen, Minimierung, PLAs

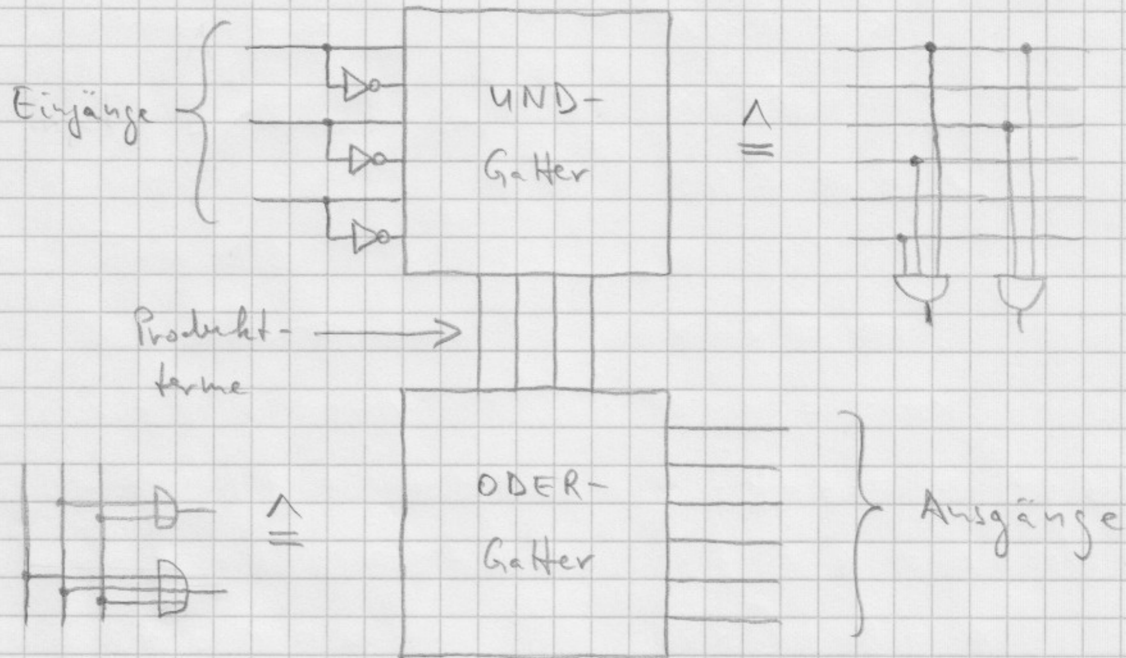
Jede Boole'sche Funktion läßt sich in den beiden "Normalformen" DNF (disjunktive Normalform, "sum-of-products") und KNF (konjunktive Normalform, "product-of-sums") darstellen. Es gibt Verfahren (Karnaugh-Diagramme, Quine-McCluskey-Algorithmus, Espresso-II), die eine minimale (oder angenähert minimale) Repräsentation berechnen.

<u>Bsp.</u>: $A \cdot B \cdot \overline{C} \cdot \overline{D} + \overline{A} \cdot B \cdot \overline{C} \cdot D + A \cdot B \cdot \overline{C} \cdot D + \overline{A} \cdot B \cdot \overline{C} \cdot \overline{D}$

$+ A \cdot B \cdot C \cdot D = B \cdot \overline{C} + A \cdot B \cdot D$

"Produktterme"

Der Aufbau eines PLA ("programmable logic array") spiegelt diese zweistufige Logik wieder:



Bem.: Ein ROM lässt sich auffassen als ein vollständig dekodiertes PLA: Alle 2^n Eingangskombinationen sind als Produktterme vorhanden. Die Programmierung des ROMs geschieht nur mittels der ODER-Gatter.

1.4.3. Don't Cares

Manchmal möchte man den Wert eines Digitalsignals nicht spezifizieren: ein sog. "don't care", bezeichnet mit "x". Don't Cares kommen an zwei Stellen vor, an denen sie verschiedene Dinge bedeuten.

Bsp.:

A	B	C	Out
0	x	0	1
0	0	1	0
0	1	1	0
1	x	0	x
1	0	1	x
1	1	1	1

a) Don't Cares in den Eingängen einer Funktion:

Die Zeile 0 x 0 | 1 ist in Wirklichkeit eine Kurzschreibweise für die beiden Zeilen

0	0	0	1
0	1	0	1

und bedeutet also, daß die Funktion von der betreffenden Variablen (hier B) an dieser Stelle nicht abhängt.

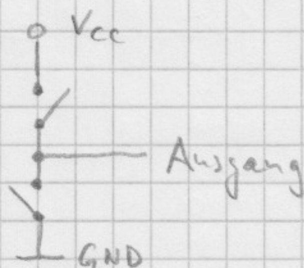
b) Don't Cares in den Ausgängen einer Funktion:

Die Zeile 1 0 1 | x bedeutet, daß der Wert der Funktion für die Eingangskombination 1 0 1 keine Rolle spielt und deshalb beliebig gewählt werden kann (z.B. so, daß die Funktion einfach realisiert werden kann).

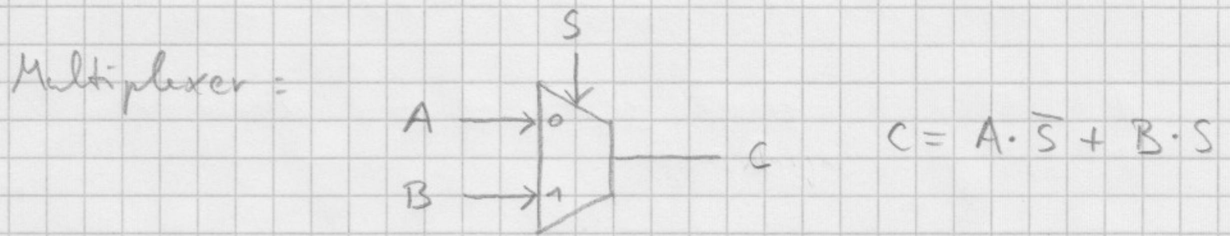
Kombinationen aus a) und b) sind möglich.

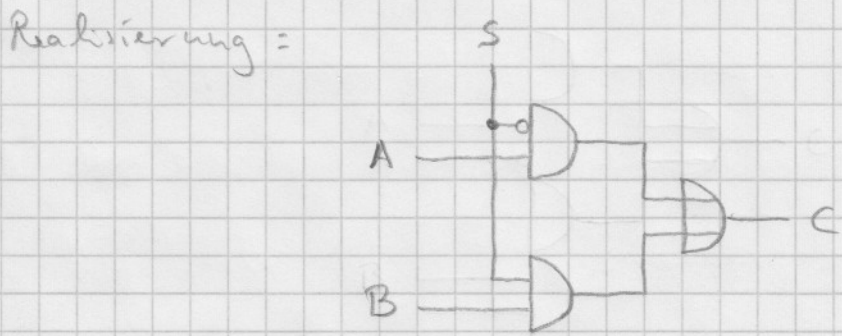
1.4.4. Tri-State-Gatter

Man kann Gatter mit einem zusätzlichen Steuereingang ("output enable") ausrüsten. In der Stellung "enabled" erlaubt er, daß das Gatter eine ϕ oder eine 1 am Ausgang erzeugt. In der Stellung "disabled" wird durch Sperren beider Transistoren (Öffnen beider Schalter) das Gatter effektiv vom Ausgang abgetrennt ("hochohmig gemacht", Ausgangswert "Hi-Z" oder kurz z).



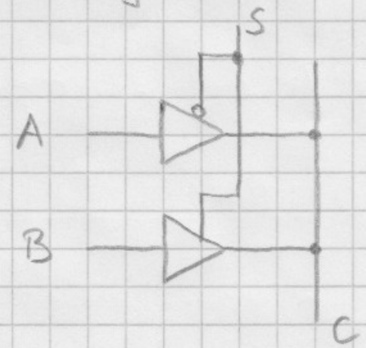
Verwendung: „Verteilter Multiplexer"





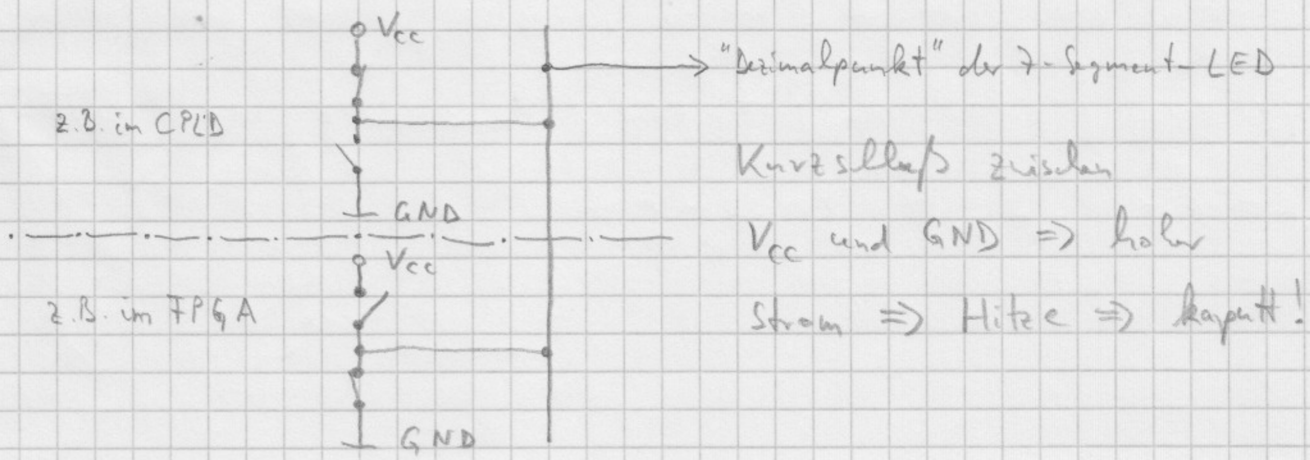
Bei 2^n Eingängen n Steuerleitungen nötig. Bei großem n hoher Schaltungsaufwand, viele Leitungen.

Realisierung mit Tri-State-Gattern:



Vorteile: Leicht erweiterbar, Logik dezentral

Nachteile: Langsamer (wg. kapazitiver Last am Ausgang), Selbstzerstörung bei falscher Ansteuerung



1.4.5. Vorsicht: Hazards!

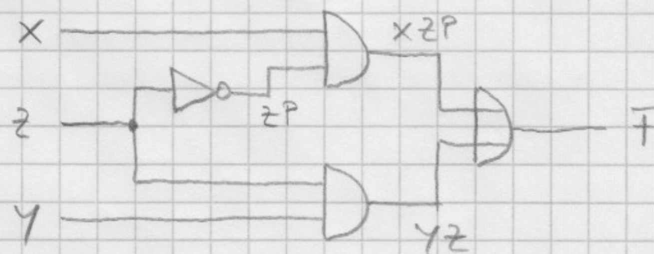
Hazard: Kurzer Puls ("Glitch") auf einem Ausgang, hervorgerufen durch Laufzeiten in der Schaltung

Hazards gibt's bei kombinatorischen Schaltungen in 3 Varianten:

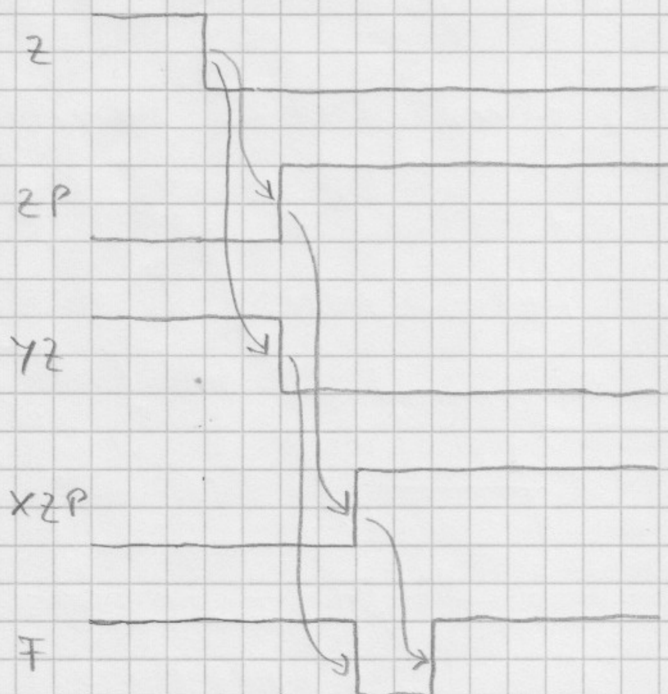
a) Statischer 1-Hazard

Ein Paar von Eingangswerte-Kombinationen, die sich nur in einer Eingangsvariablen unterscheiden und beide 1 am Ausgang erzeugen, so daß am Ausgang ein kurzer ϕ-Impuls erscheint.

Bsp.:



$X = 1,\ Y = 1,\ Z = 1 \rightarrow \phi$



b) Statischer ϕ-Hazard

Das ist das duale Gegenstück zu a).

c) Dynamischer Hazard

Ein Ausgang ändert sich mehr als einmal, obwohl sich nur ein Eingang einmal geändert hat.

Das kann passieren, wenn es mehrere Pfade vom Eingang zum Ausgang gibt, mit unterschiedlichen Verzögerungszeiten.

Wahrheiten über Hazards:

1. Hazards in ad-hoc-Schaltungen sind schwer zu finden.
2. Zweistufige Logik weist keine dyn. Hazards auf.
3. Zweistufige UND-ODER-Logik hat keine stat. ϕ-Hazards, kann aber stat. 1-Hazards haben.
4. Zweistufige ODER-UND-Logik hat keine stat. 1-Hazards, kann aber stat. ϕ-Hazards haben.

Vor.: Niemals liegt x und $\overline{x}$ am gleichen Gatter an! (→ 2., 3., 4.)

5. Stat. Hazards lassen sich mit Karnaugh-Diagrammen finden und beheben. Unser Bsp. von oben: $F = X\overline{Z} + YZ$

Z \ XY	00	01	11	10
0			1	1
1		1	1	

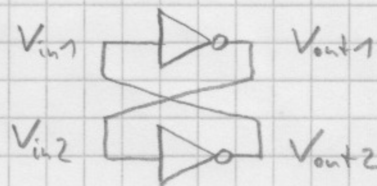
Hinzunahme des "Konsens-Terms" XY behebt den Hazard; $F = X\overline{Z} + YZ + XY$ ist frei von Hazards.

6. Hazards sind extrem wichtig beim Entwurf von nichtsynchronen sequentiellen Schaltungen. Sie spielen keine Rolle in synchronen Schaltwerken: Die Ausgänge von Schaltnetzen werden nur an Taktflanken ausgewertet, d.h. wenn alle Hazards abgeklungen sind.

1.5. Sequentielle Logik

1.5.1. Bistabile Elemente, Metastabilität

Schaltung:

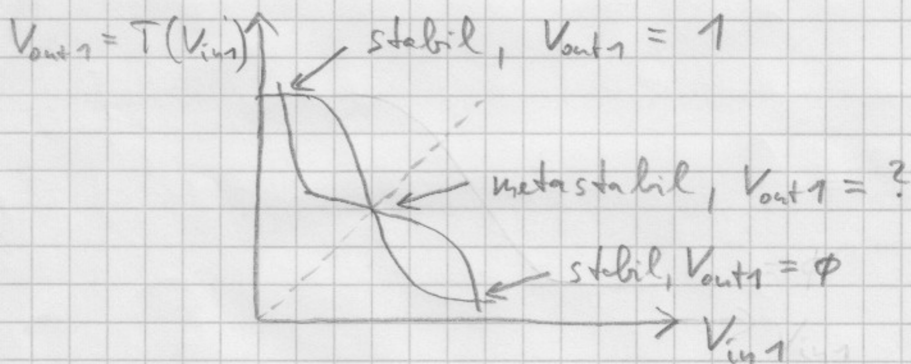


Analyse:

$$V_{out1} = T(V_{in1}) \quad , \quad V_{out2} = T(V_{in2})$$

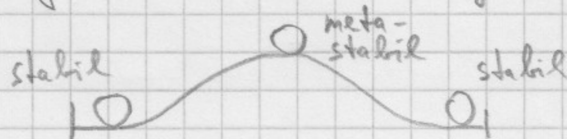
$$\Rightarrow V_{in1} = V_{out2} = T(V_{in2}) = T(V_{out1}) = T(T(V_{in1}))$$

$$\Rightarrow T^{-1}(V_{in1}) = T(V_{in1})$$



Bem.:

1. Jede sequentielle Schaltung kann in metastabile Zustände gebracht werden – nicht nur beim Einschalten der Betriebsspannung, sondern auch beim Verletzen der Timing-Bedingungen (mehr in Kürze). Verweildauer: unbegrenzt!
2. Verlassen des metastabilen Zustandes durch Rauschen: ist der Gleichgewichtspunkt erst mal ein wenig verlassen, wirkt die Rückkopplung (exponentiell) verstärkend.
3. Analogon: "Ball und Hügel"

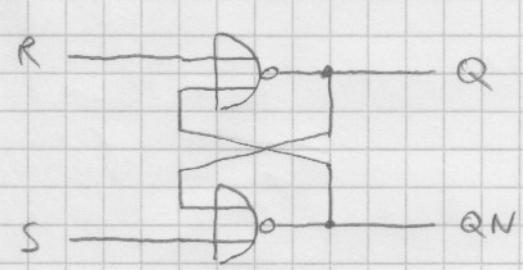


1.5.2. Latches und Flip-Flops

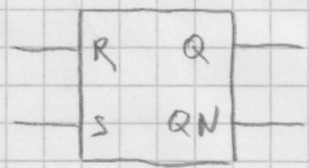
Flip-Flop: sequentielle Schaltung, die ihre Ausgänge nur zu best. Zeiten ändert, gegeben durch "Takt"

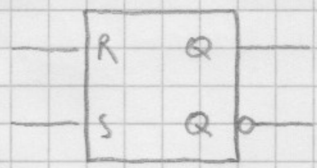
Latch: sequentielle Schaltung, die kontinuierlich auf Eingangssignaländerungen reagiert

1.5.2.1. S-R-Latch



S	R	Q	QN
0	0	letztes Q	letztes QN
0	1	0	1
1	0	1	0
1	1	0	0





↖ nicht ganz korrekt f. $S = R = 1$

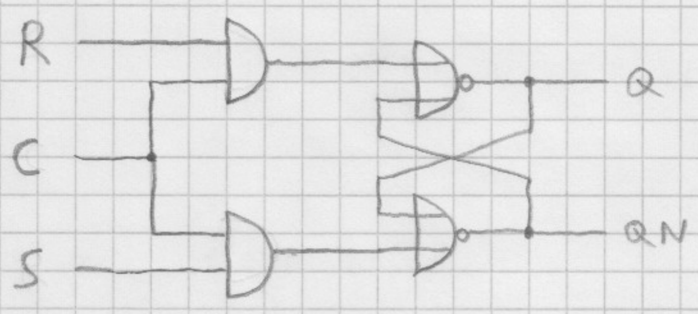
Das S-R-Latch kann metastabil werden, wenn die Pulsdauer an R oder S zu kurz ist, oder wenn bei $S = R = 1$ die Übergänge $R: 1 \rightarrow 0$ und $S: 1 \rightarrow 0$ zeitlich zu nahe beieinander liegen. Dann ist sogar "Schwingen" der Ausgänge möglich:

Q $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \ldots$

QN $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \ldots$

1.5.2.2. S-R-Latch mit Enable

"Enable" steuert, ob das Latch auf seine Eingänge reagiert.



S	R	C	Q	QN
0	0	1	letztes Q	letztes QN
0	1	1	0	1
1	0	1	1	0
1	1	1	0	0
x	x	0	letztes Q	letztes QN

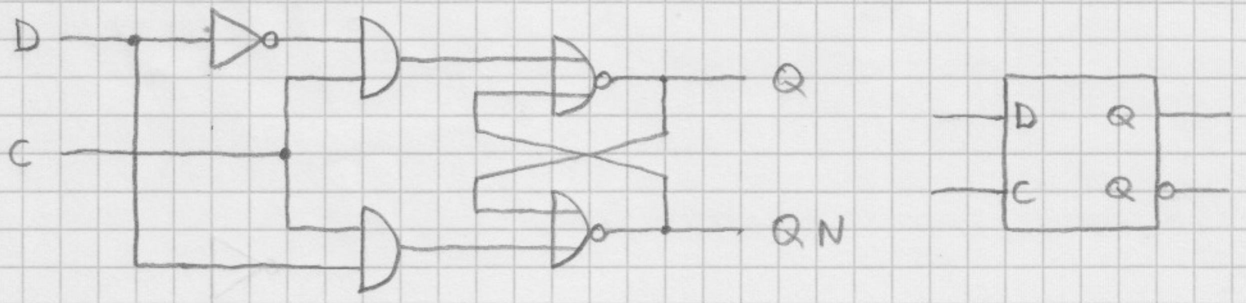
Metastabilität: bei $S = R = C = 1$ $C: 1 \rightarrow 0$

oder: bei $S = R = C = 1$ $S: 1 \rightarrow 0$ und $R: 1 \rightarrow 0$

oder: zu kurze Pulsdauer bei S, R oder C

1.5.2.3. D-Latch

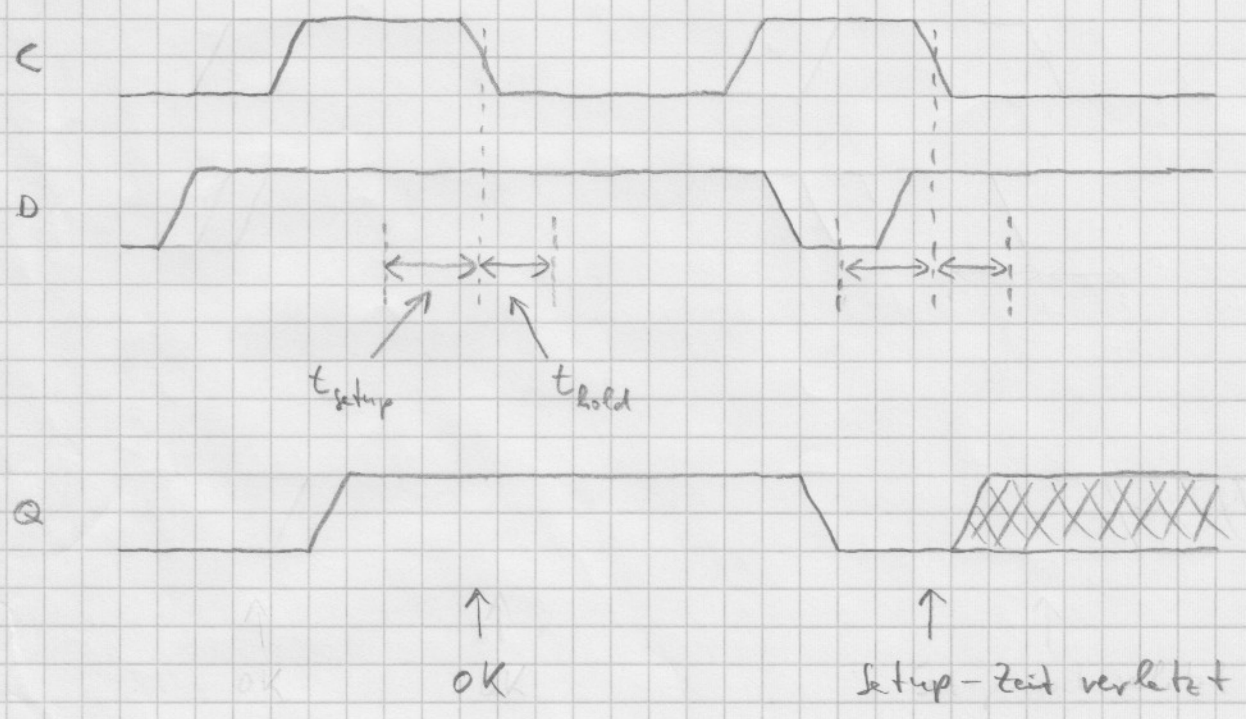
Speichert ein Datum (1 Bit). Dazu wird $S = \overline{R} = D$ gesetzt:



C	D	Q	QN	
1	0	0	1	Latch ist offen ("transparent")
1	1	1	0	
0	x	letztes Q	letztes QN	Latch ist geschlossen ("hält")

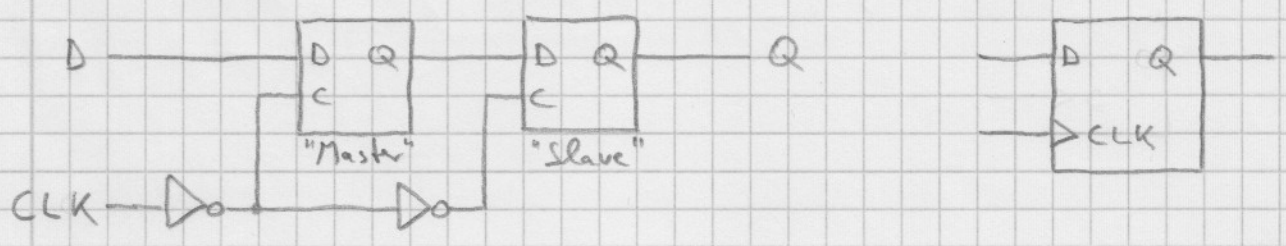
Metastabilität: kritisch ist die fallende Flanke von C

In einem best. "Fenster" um diese Flanke herum darf sich der Eingang D <u>nicht ändern</u>, sonst droht Metastabilität!



1.5.2.4. Taktflankengesteuertes ("edge triggered") D-Flip-Flop

Schaltung, die den D-Eingang nur auf einer Flanke des Takt-Eingangs speichert. Besteht aus zwei D-Latches, die mit gegenphasigem Takt angesteuert werden:



Wirkungsweise: Während CLK = 0 ist der Master transparent; sein Ausgang folgt dem Dateneingang. Der Slave hält aber den vorigen Wert. Wird CLK = 1, dann schließt der Master und der Slave übernimmt diesen Wert. Während der ganzen Zeit mit CLK = 1 folgt der Slave dem Master; dessen Wert ändert sich aber nicht mehr, weil er geschlossen ist.

D	CLK	Q
0	↑	0
1	↑	1
x	0	letztes Q
x	1	letztes Q

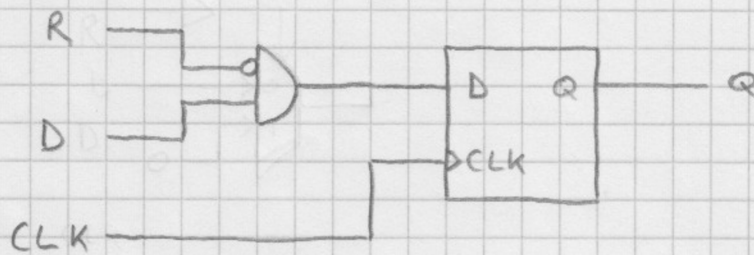
<u>In Verilog:</u>

```
always @ (posedge CLK) begin
    Q <= D;
end
```

Metastabilität: Genau wie beim D-Latch sind Setup- und Holdzeiten in Bezug auf die aktive Flanke des Taktsignals (↑) zu beachten.

<u>Bem.:</u> D-Flip-Flops können mit zusätzlichen asynchronen Setz- und Rücksetzeingängen ausgestattet sein, die es erlauben, das Flip-Flop unabhängig vom Takt auf 1 oder 0 zu setzen ("preset" und "clear"). Diese sollten in synchronen Schaltungen nicht verwendet werden (außer bestenfalls zum Herstellen eines bekannten Start-Zustandes beim Reset — selbst da ist Vorsicht geboten).

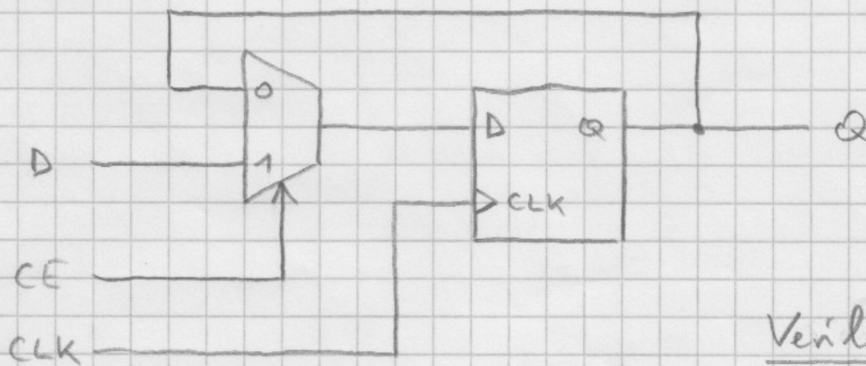
1.5.2.5. D-Flip-Flop mit synchronem Reset



Verilog: siehe ①

1.5.2.6. D-Flip-Flop mit Taktfreigabe ("clock enable")

WICHTIG: Entgegen dem deskriptiven Namen "Taktfreigabe" hat das mit dem Takt nicht das geringste zu tun!!



Verilog: siehe ②

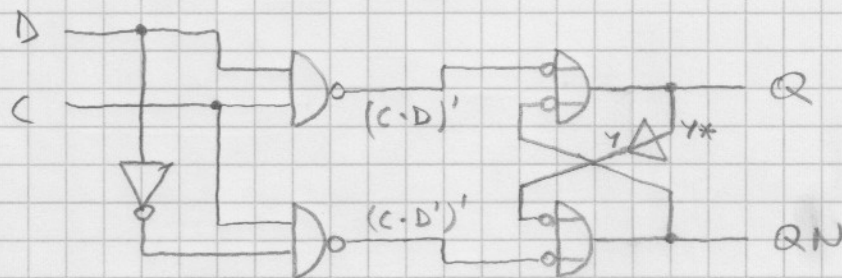
---

①
```
always @ (posedge CLK) begin
    if (R)
        Q <= Φ;
    else
        Q <= D;
end
```

②
```
always @ (posedge CLK) begin
    if (CE)
        Q <= D;
end
```

1.5.3. Analyse sequentieller Schaltungen

Ist i. A. extrem aufwendig. Wir behandeln ein Bsp.:



<u>1. Schritt:</u> Aufbrechen der Rückkopplungen durch fiktive Puffer (hier ist <u>ein</u> Puffer erforderlich). Die Ausgänge der Puffer sind die "Zustandsvariablen" (hier: Y). Die "nächsten" Werte der Zustandsvariablen werden mit einem "*" versehen. Dann läßt sich der nächste Zustand aus dem jetzigen Zustand und den Eingangsvariablen ausdrücken ("Anregungsfunktion"):

$$Y* = (C \cdot D) + (Y' + C \cdot D')'$$

$$= C \cdot D + Y \cdot C' + Y \cdot D$$

<u>2. Schritt:</u> Aufstellen der Übergangstabelle für Y*:

Y \ CD	00	01	11	10
0	0	0	1	0
1	1	1	1	0

Im Allgemeinen hat eine solche Tabelle 2^n Zeilen für n Rückkopplungsschleifen und 2^m Spalten für m Eingänge.

Zunächst Annahme: Schaltung wird im "Grundmodus" betrieben,

d.h. es wird zu einem gegeb. Zeitpunkt nur <u>ein</u> Eingang geändert und die Schaltung hat danach genügend Zeit, in einen stabilen Zustand überzugehen.

"Totaler Zustand": Kombination aus "internem Zustand" (= Werte in den Rückkopplungsschleifen) und "Eingangszustand" (= Werte der Eingangsvariablen). "Stabiler totaler Zustand": Kombination so, daß nächster interner Zustand gleich jetzigem internem Zustand ist.

<u>3. Schritt</u>: Aufstellen der Zustands- und Ausgangstabelle

Wir benennen die internen Zustände (hier: S0, S1), markieren die stabilen totalen Zustände und schreiben die Werte der Ausgangsgrößen mit in eine Tabelle:

$$Q = C \cdot D + C' \cdot Y + D \cdot Y$$

$$QN = C \cdot D' + Y'$$

S \ CD	00	01	11	10
S0	(S0), 01	(S0), 01	S1, 11	(S0), 01
S1	(S1), 10	(S1), 10	(S1), 10	S0, 01

Nun kann man Übergänge untersuchen, z.B. S0/00, D→1. Dann gelangt man zum stabilen totalen Zustand S0/01, d.h. interner Zustand (und Ausgänge) bleiben gleich. Wechselt jetzt C→1, kommt man in den instabilen totalen Zustand S0/11 und von dort in den stabilen totalen Zustand S1/11 (Ausgänge: 10).

4. Schritt: Analyse des Verhaltens bei "gleichzeitiger" Veränderung an den Eingängen. Wirklich gleichzeitig gibt's nicht – wir müssen bei m "gleichzeitig" wechselnden Eingängen jede der m! verschieden Reihenfolgen gesondert betrachten! Bsp.: Start in S1/11, CD → 00

S \ CD	00	01	11	10
S0	(S0), 01	(S0), 01	S1, 11	(S0), 01
S1	(S1), 10	(S1), 10	(S1), 10	S0, 01

Wenn C zuerst wechselt: S1/11 → S1/01 → S1/00

Wenn D zuerst wechselt: S1/11 → S1/10 → S0/10 → S0/00

⇒ Zustand unbekannt!

Aber: Nicht jeder "gleichzeitige" Wechsel von Eingangssignalen muß unvorhersagbares Verhalten zur Folge haben!

Bsp.: S0/00, CD → 11

Beide Übergangssequenzen enden bei S1/11.

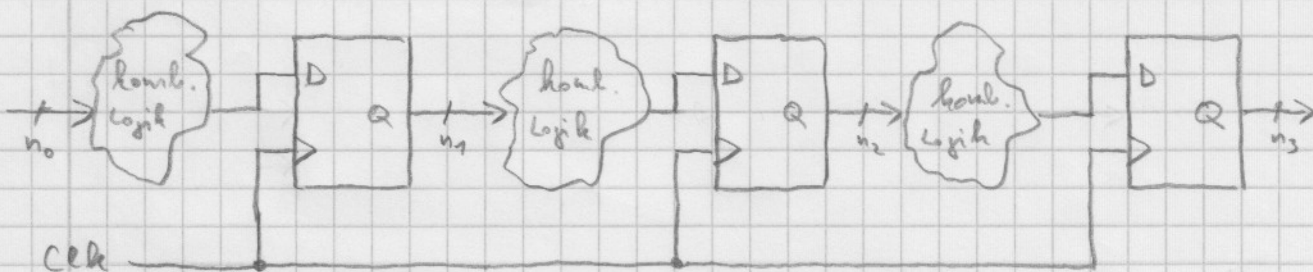
Fazit: Beliebige sequentielle Schaltungen sind nur mit enormem Aufwand zu analysieren, und noch schwieriger zu konstruieren! Ausweg: Synchrone Schaltwerke, in denen die sequentiellen Schaltungen auf Speicherglieder beschränkt werden, die gut verstanden sind (D-Latches bzw. D-Flip-Flops).

1.6. Synchrone Schaltwerke und Automaten

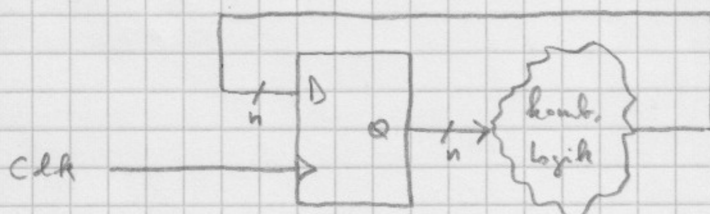
1.6.1. Prinzip und Realisierung

<u>Prinzip</u>: Sequentielle Schaltwerksteile sind <u>nur</u> in speichernden Elementen zu finden. Diese sind alle von derselben Sorte (z.B. positiv-flankengesteuerte D-Flip-Flops) und werden alle vom selben Takt angesteuert.

<u>Vorteil</u>: Das Einhalten der Setup-Zeiten kann immer durch Senken der Taktfrequenz erreicht werden; Hazards und Durchlaufzeiten der kombinatorischen Schaltnetze wirken sich höchstens begrenzend auf die Taktfrequenz aus, machen die Schaltung aber nicht funktionsunfähig!



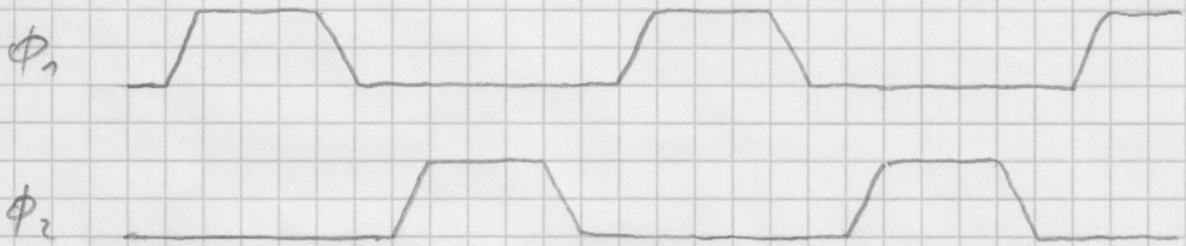
<u>Beachte</u>: Jedes Speicherelement wird <u>gleichzeitig</u> gelesen und (mit der Taktflanke) geschrieben! Also ist auch folgende Anordnung zulässig ("synchroner Automat"):



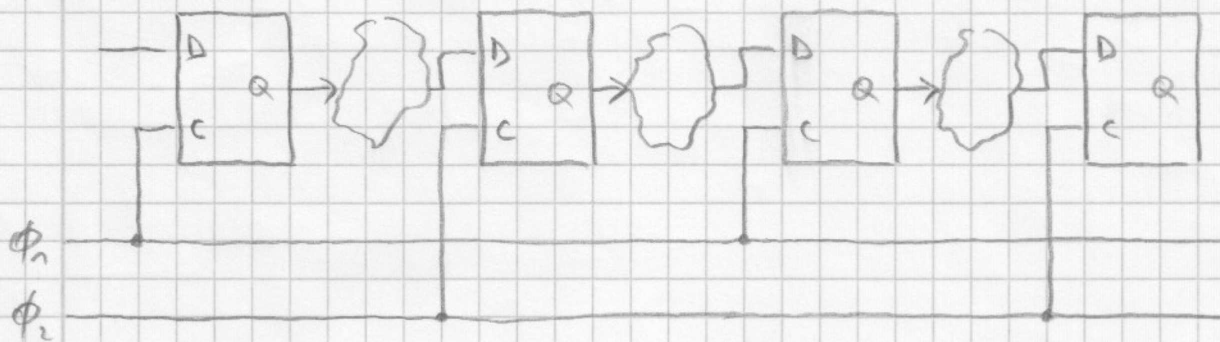
Der Wert von Q heißt "Zustand", der Wert, der gleichzeitig an D anliegt "Folgezustand" des Automaten.

Die komb. Logik berechnet also den Folgezustand aus dem Zustand.

<u>Bem.</u>: Man kann anstelle von D-Flip-Flops auch D-Latches benutzen, muß allerdings die oben gezeigte Schaltung modifizieren, da sonst bei CLK = 1 (Latches sind transparent) kein Zustand gehalten werden könnte. Abhilfe schafft ein "nichtüberlappender Zweiphasen-Takt":



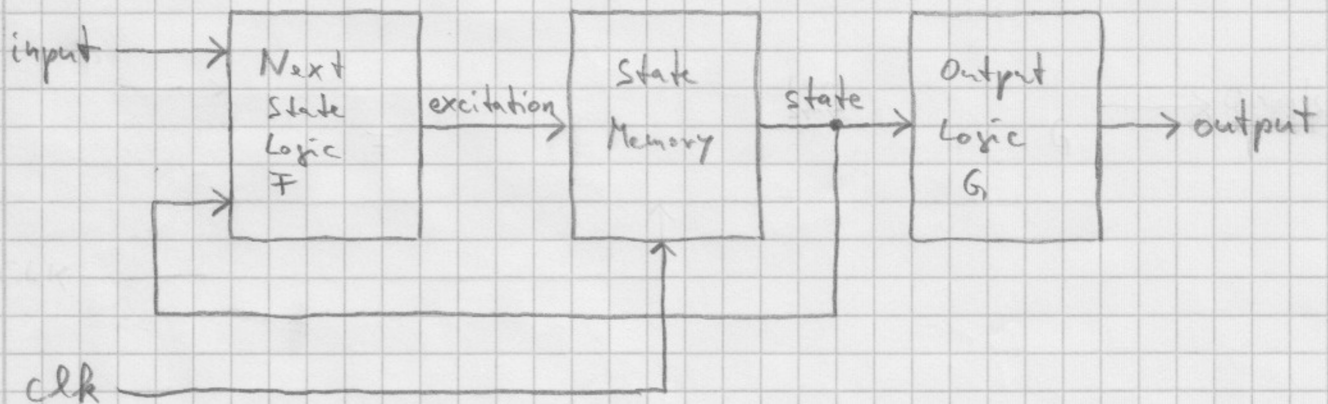
Dann müssen die speichernden Elemente abwechselnd mit Φ_1 und Φ_2 getaktet werden:



Beachte: Wenn man jedes zweite Schaltnetz wegläßt und die beiden Latches zusammenfaßt, erhält man wieder die D-Flip-Flop-Variante.

1.6.2. Moore-Automat

Im Allgemeinen wird ein Automat auch Eingänge besitzen, mit denen er etwas über seine Umgebung erfährt, und Ausgänge, mit denen er seine Umgebung beeinflußt.
Je nachdem, was die Eingänge steuern bzw. wie die Ausgänge berechnet werden, unterscheidet man verschiedene Automaten.
Beim Moore-Automaten hängt die Ausgabe nur vom Zustand ab, nicht von der Eingabe:

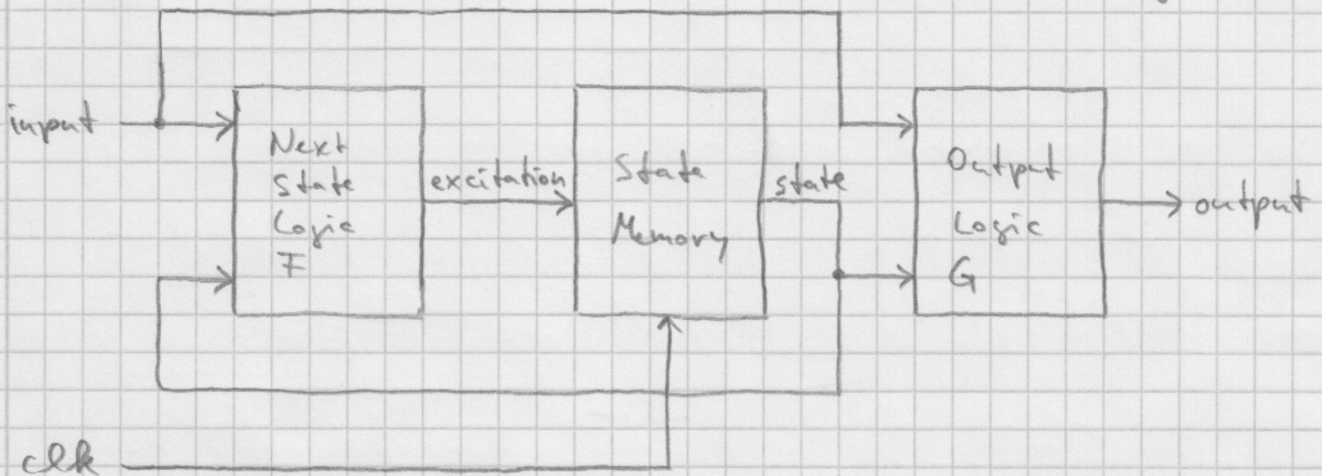


In Verilog:

```
assign output = G(state);
always @(posedge clk) begin
    case (state)
        :
        n: begin
            if (input == ...) begin
                state <= m;   // excitation = F(state, input)
            end else
            :
        end
        :
    endcase
end
```

1.6.3. Mealy-Automat

Hier hängt die Ausgabe (asynchron) auch von der Eingabe ab:



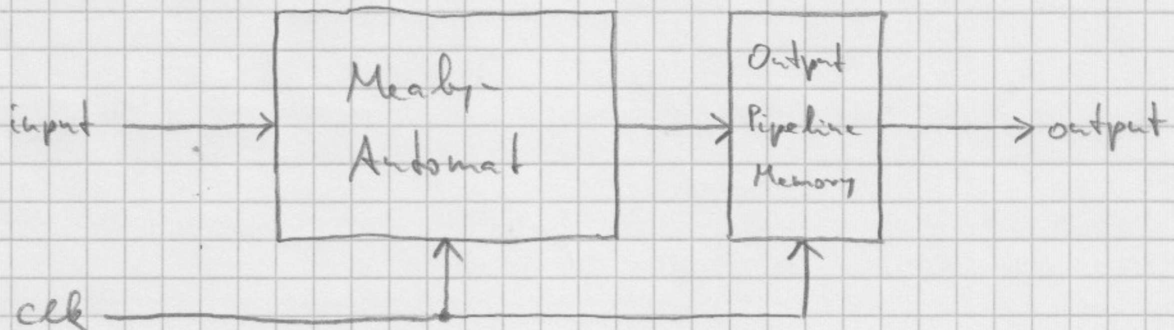
In Verilog: Wie beim Moore-Automaten, aber mit

```
assign output = G(state, input);
```

Vergleich Moore/Mealy: Moore-Automaten sind potentiell schneller als Mealy-Automaten, haben aber möglicherweise mehr Zustände als diese.

1.6.4. Mealy-Automat mit synchronem ("Pipeline"-) Ausgang

Dieser Automat entsteht aus dem Mealy-Automaten durch "Anhängen" einer Speicherstufe an die Ausgänge:



In Verilog:

```
always @(posedge clk) begin
   case (state)
        :
        n: begin
             if (input == ...) begin
                 state <= m;
                 output <= x;
             end else
                 :
           end
        :
   endcase
end
```

Bem.: Diese Variante ist sehr schnell, man muß aber die Ausgangswerte schon einen Taktzyklus früher berechnen können.
(Persönliche Anmerkung: Achtung, Hirnsausen!)

① →

1.6.6. Asynchrone Eingänge und Synchronisierer

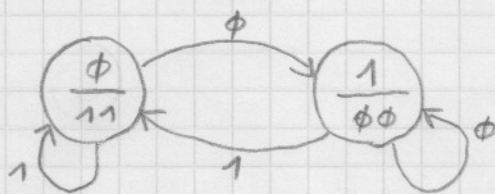
Nicht alle Teilsysteme eines Rechners werden mit dem gleichen Takt getaktet (wie drückt man synchron zu 1GHz eine Taste?). Man benötigt "Synchronisierer", die ein asynchrones Eingangssignal synchron zu einem gegb. Takt abtasten. Es ist normalerweise nicht schlimm, wenn der korrekte Wert erst einen Takt später übernommen wird, aber es gibt zwei Fehler, die man hier leicht macht:

1. Synchronisation des gleichen Signals an mehreren Stellen

1.6.5. Zur Konstruktion von Automaten ("FSMs")

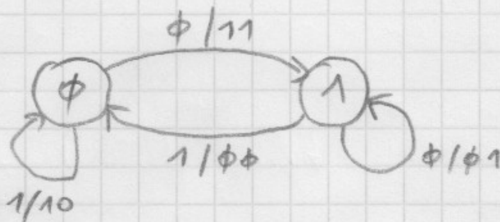
1. Stehen die Ausgaben an den Zuständen: Moore-Automat

Bsp.:



2. Stehen die Ausgaben an den Übergängen: Mealy-Automat

Bsp.



3. Häufig hat man beide Sorten Ausgänge gleichzeitig (formal ist das dann ein Mealy-Automat).

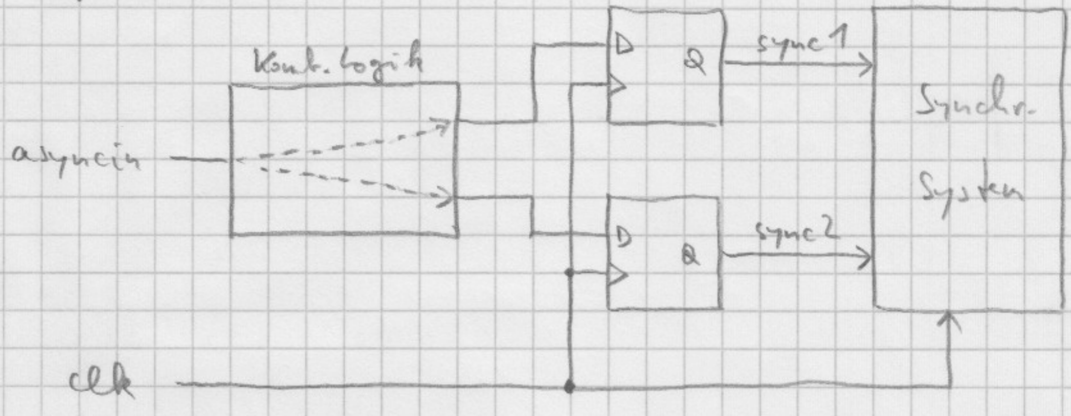
4. Minimierung der Zustände: ist machbar, aber ein vernünftiges Design macht das i.d.R. überflüssig.

5. Optimale Kodierung der Zustände: ist abhängig von der verwendeten Technologie → die Synthesewerkzeuge können das erledigen. Tatsächlich wird häufig eine "One-Hot-Kodierung" gewählt: n Zustände werden durch n Bits (nicht ld n Bits!) repräsentiert, von denen genau eines 1, der Rest Φ ist.

6. Niemals vergessen: FSMs müssen einen Reset-Eingang haben (sonst funktioniert z.B. One-Hot-Kodierung nicht)!

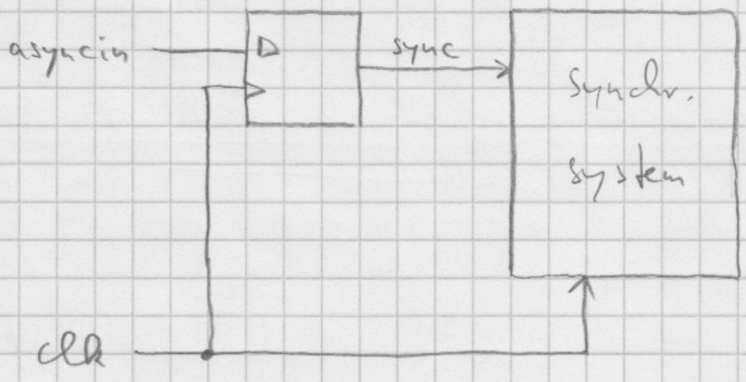
7. Ein gutes Design sieht Übergänge von "unerreichbaren" Zuständen zu einem Fehlerzustand oder zum Resetzustand vor!

Das Problem tritt insbesondere auf, wenn die Mehrfachsynchronisation durch kombinatorische Logik verdeckt wird:



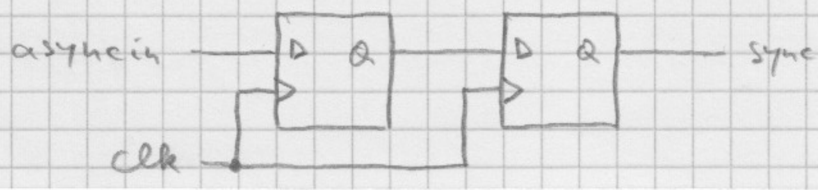
Hier ist es sehr wahrscheinlich, daß bei einer Änderung von asyncin das eine Flip-Flop den alten Wert aus der komb. Logik, das andere aber den bereits geänderten Wert erkennt.

Richtige Vorgehensweise: <u>zuerst</u> (vor irgendwelcher Logik) und <u>nur an einer Stelle</u> synchronisieren!



2. Unterschätzen der Gefahr von Metastabilität

Beim Erhöhen der Taktrate steigt die Gefahr von Metastabilität exponentiell an! Deshalb sollten zwei (in Spezialfällen drei) Synchronisierstufen vorgesehen werden:



2. Rechnerarchitektur

Die Rechnerarchitektur versucht, Gesetze zu finden und Strukturen zu beschreiben, die zum Bau von Rechnern mit hoher Leistung ("Performance") führen.

2.1. Performance

Zwei gebräuchliche Metriken:

Als einzelner Benutzer ist man intressiert an der "Antwortzeit".

Als Computer-Center-Manager ist man am "Durchsatz" intressiert.

Verkürzung der Antwortzeit verbessert i. A. auch den Durchsatz.

Im Folgenden wird die Antwortzeit ("Execution time") zur Performance in Bezug gesetzt:

$$\text{Performance} = \frac{1}{\text{Execution time}}$$

~~Bem.: Beim Messen der Ausführungszeit ist zu unterscheiden zwischen "User CPU time", "System CPU time" und "Elapsed time". Hier geht's um die "User CPU time".~~

2.1.1. Die Performance-Gleichung

$$\text{Execution time} = \frac{\text{Clock cycles}}{\text{Program}} \cdot \frac{\text{Seconds}}{\text{Clock cycle}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Clock cycle}}$$

↑ "CPI", dient zum Vergleich verschiedener Implementierungen des gleichen Befehlssatzes

Der CPI ist ein gewichtetes Mittel:

$$CPI = \sum_{i \in \text{Instrs}} CPI_i \cdot p_i$$

(p_i: rel. dyn. Häufigkeit der Instr. i)

<u>Bsp.:</u> Maschine A: Clock cycle time = 1 nsec, CPI = 2.0

Maschine B: " " " = 2 nsec, CPI = 1.2

Wie ist das Performance-Verhältnis der beiden Maschinen?

Die Zahl der im Progr. ausgeführten Instruktionen sei n.

$$\text{Execution time}_A = n \cdot 2.0 \cdot 1\,\text{nsec} = 2n\ \text{nsec}$$

$$\text{Execution time}_B = n \cdot 1.2 \cdot 2\,\text{nsec} = 2.4n\ \text{nsec}$$

$$\Rightarrow \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Ex.time}_B}{\text{Ex.time}_A} = \frac{2.4n}{2n} = 1.2$$

<u>Bem.:</u> Komplexere Instruktionen führen zwar zu einer kleineren Anzahl von ausgeführten Instruktionen, erhöhen aber i.d. Regel die Clock cycle time oder den CPI.

$\Rightarrow$ Beim Vergleich von Rechnern <u>alle drei Faktoren</u> bedenken!

2.1.2. Programme, um Performance zu messen

Sog. "Benchmarks" dienen zum Vergleich von Rechnern.

Fazit aus vielen Jahren mit fehlerhaft entworfenen, geschickt ausgetricksten, etc, etc, Benchmarks lassen nur einen Schluß zu: Die zu erwartende Performance muß an realen Programmen, am besten der eigenen Applikation, gemessen werden.

2.1.3. Amdahl's Gesetz

Trugschluß: Man erwartet, daß eine Beschleunigung eines Aspekts einer Maschine die Performance derselben um einen Betrag proportional zur Beschleunigung steigert.

Richtig ist: Execution time (mit Beschleunigung)

$$= \left(\frac{\text{Execution time (betroffen von Beschleunigung)}}{\text{Faktor der Beschleunigung}} + \text{Execution time (nicht betroffen von Beschleunigung)} \right)$$

Bsp.: Ein Programm läuft 100 sec auf einer best. Maschine, davon werden 80 sec für Multiplikationen verbraucht. Um welchen Faktor müssen Multiplikationen beschleunigt werden, so daß das Programm danach 5 mal so schnell läuft?

$$\text{Exec. time (mit Beschl.)} = \frac{80\text{ sec}}{n} + 20\text{ sec} \overset{!}{=} \frac{100\text{ sec}}{5}$$

$\Rightarrow \frac{80\text{ sec}}{n} = 0 \Rightarrow$ keine (endl.) Beschleunigung der Multiplikationen kann eine 5-fache Performance der Maschine bewirken, wenn Multiplikationen nur 80% der ausgeführten Instruktionen ausmachen!

$\Rightarrow$ Amdahl's Gesetz: "Speedup" $= \frac{\text{Performance mit Beschl.}}{\text{Performance ohne Beschl.}}$

$$= \frac{\text{Exec. time (ohne)}}{\text{Exec. time (mit)}} = \frac{1}{(1-p) + \frac{p}{s}}$$

mit $p = \frac{\text{Exec. time (betroffen v. Beschl.)}}{\text{Exec. time (ohne Beschl.)}}$, s = Faktor der Beschl.

2.2. Instruktionssatz-Architektur

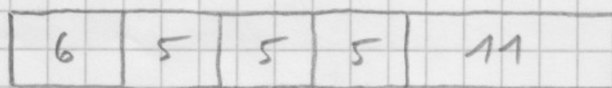
Das ist eine der wichtigsten Abstraktionen im Rechner: das Interface zwischen Hardware und Software. Für ein- und dieselbe ISA sind viele Implementierungen möglich, mit sehr verschiedener Performance (und Kosten).

Bei uns: "ECO32e", eine Untermenge des MIPS-Befehlssatzes (mit anderer Kodierung), 34 Befehle, jeder belegt genau 1 Wort.

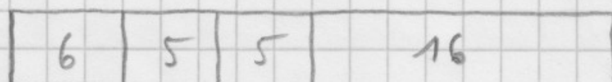
2.2.1. Befehlsformate

Beschreiben die Einteilung der Befehle in Klassen und die entspr. Zuordnung von Bits in den Befehlen zu "Feldern", die zur Kodierung der Befehlsbestandteile dienen.

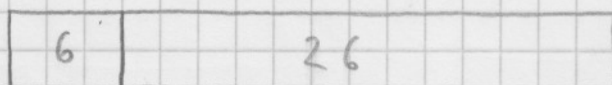
R-Format:



I-Format



J-Format



Festlegung des Formats durch den Opcode!

Bem.: Es werden verschiedene kleinere Variationen der Hauptformate benutzt, z.B.:

- bei Arithmetik-Befehlen wird der Immediate-Operand vorzeichenrichtig aufgeweitet ("sign-extended"), bei Logik-Befehlen aber mit Nullen aufgefüllt ("zero-extended")
- zum Berechnen einer Datenadresse wird der Immediate-Operand so verwendet, wie er vorliegt, zum Berechnen einer Instruktionsadresse (bei Sprüngen) aber erst um zwei Bits nach links geschoben (da Instruktionen nur an durch 4 teilbaren Adressen liegen)

2.2.2. Befehlssatz

Im ECO32e haben wir Befehle für

- Arithmetik (+, −)
- Logik (&, |, xor, xnor)
- bedingte Sprünge ("branches")
- unbedingte Sprünge ("jumps")
- Unterprogrammsprünge ("jump and link")
- Datentransfer vom ("load") und zum ("store") Speicher in den drei Transferbreiten Word, Halfword, Byte

Austeilen "ECO32e Architecture"

2.3. Implementierung des Befehlssatzes

2.3.1. Strategien

Es gibt drei Strategien, um einen gegeb. Befehlssatz zu realisieren:

a) Single-Cycle: jede Instruktion benötigt genau einen Taktzyklus zur Ausführung

Nachteile: Die Zyklusdauer muß sich nach dem kompliziertesten (längsten) Befehl richten; Hardware (wie z.B. Addierer) kann nicht für verschiedene Zwecke ($PC \leftarrow PC + 4$, $RES \leftarrow OP1 + OP2$) innerhalb der gleichen Instruktion wiederverwendet werden, sondern muß mehrfach vorhanden sein; getrennte Speicher für Instruktionen und Daten erforderlich.

Abschätzung des längsten Pfades (komplizierteste Instruktion ist ldw):

- Zugriff auf Instruktionsspeicher
- Zugriff auf Register-File (Lesen)
- ALU (zur Berechnung der effektiven Adresse)
- Zugriff auf Datenspeicher
- Zugriff auf Register-File (Schreiben)

$\Rightarrow$ Obwohl CPI = 1 ist, ist die Performance schlecht, da die lange, wegen ldw notwendige Taktzeit von vielen Befehlen nicht gebraucht wird!

b) Multi-Cycle: jede Instruktion wird in eine unterschiedliche Anzahl etwa gleich langer Teilaufgaben aufgespalten. Die längste der Teilaufgaben bestimmt die Zyklusdauer; der CPI ist für versch. Befehle verschieden.
Vorteile: Ein gemeinsamer Speicher für Instruktionen und Daten; Wiederverwendung von Hardware für versch. Teilaufgaben innerhalb einer Instruktion; gute Anpassung der Instruktionsdauer an die wirklich benötigte Zeit. Zusätzlich braucht man:

1) Ein paar wenige Register, die die Ergebnisse eines Teilschrittes einer Instruktion an die nächsten Teilschritte weitergeben.

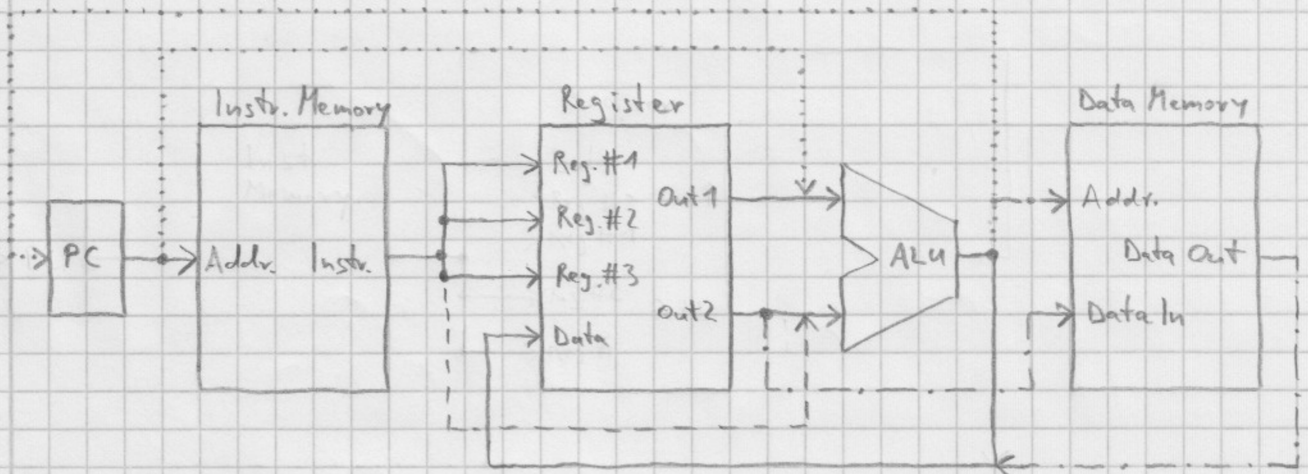
2) Ein paar wenige Multiplexer, die Eingangssignale an Bausgruppen auswählen.

3) Einen Automaten ("Steuerwerk"), der die Einzelschritte für jede Instruktion ablaufen läßt und dabei Steuersignale für den "Datenpfad" erzeugt.

c) Pipeline: Das ist eine Kombination aus a) und b). Jede Instruktion wird in eine Anzahl gleich langer Teilaufgaben zerlegt, aber jede Teilaufgabe erhält eigene Hardware. Also können bei n Teilschritten n verschiedene Teile der Hardware ("Pipeline-Stufen") parallel arbeiten – allerdings auf n unterschiedlichen Instruktionen. Damit wird zwar die Ausführungszeit <u>einer</u> Instruktion nicht verringert, aber der "Durchsatz" von Instruktionen (= ausgeführte Instruktionen pro Zeiteinheit) auf das n-fache erhöht (bestenfalls).

2.3.2. Eine Multi-Cycle-Implementierung

2.3.2.1. Abstraktes Blockschaltbild

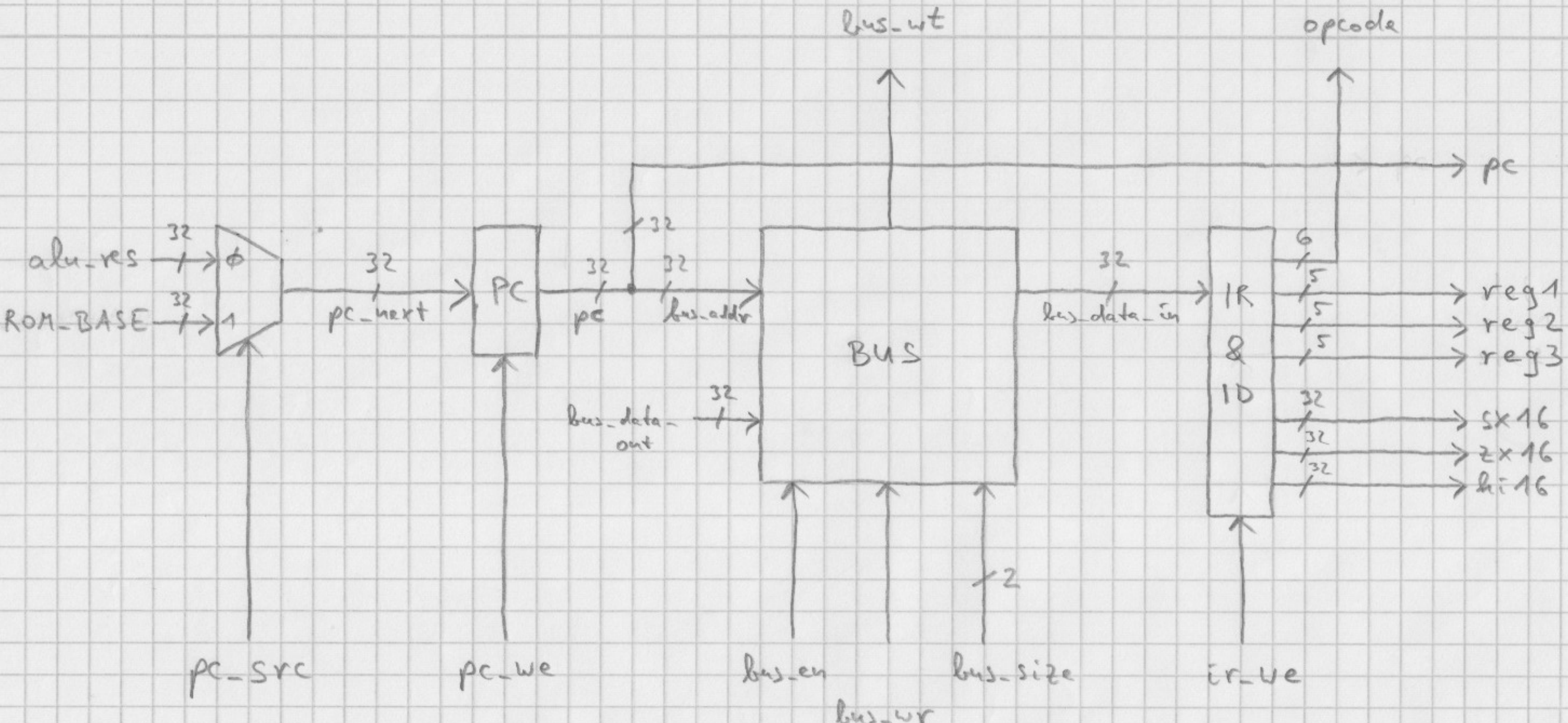


——— Register-Instruktionen
– – – Immediate-Instruktionen
······ Sprünge
–·–· Load/Store-Instruktionen

2.3.2.2. Wo werden "Auffangregister" plaziert?

Das richtet sich ganz danach, in welche Teilschritte man die Instruktionen zerlegt. Dies wiederum hängt von der Durchlaufzeit der Signale durch die Teilwerke und von der beabsichtigten Taktrate ab. Eine grobe Abschätzung für unseren FPGA ergibt: Bei 50 MHz hat man recht viel Zeit, kann also viel in einem Taktzyklus erledigen und braucht nur wenige Register. Das wird unterstützt durch die Tatsache, daß das Block-RAM zur Realisierung des Register-Files getaktet ist. Eine Ausnahme bildet der Bus – der kann langsam sein. Also werden hier Auffangregister benötigt.

2.3.2.3. Datenpfad für Instruction-Fetch/Decode

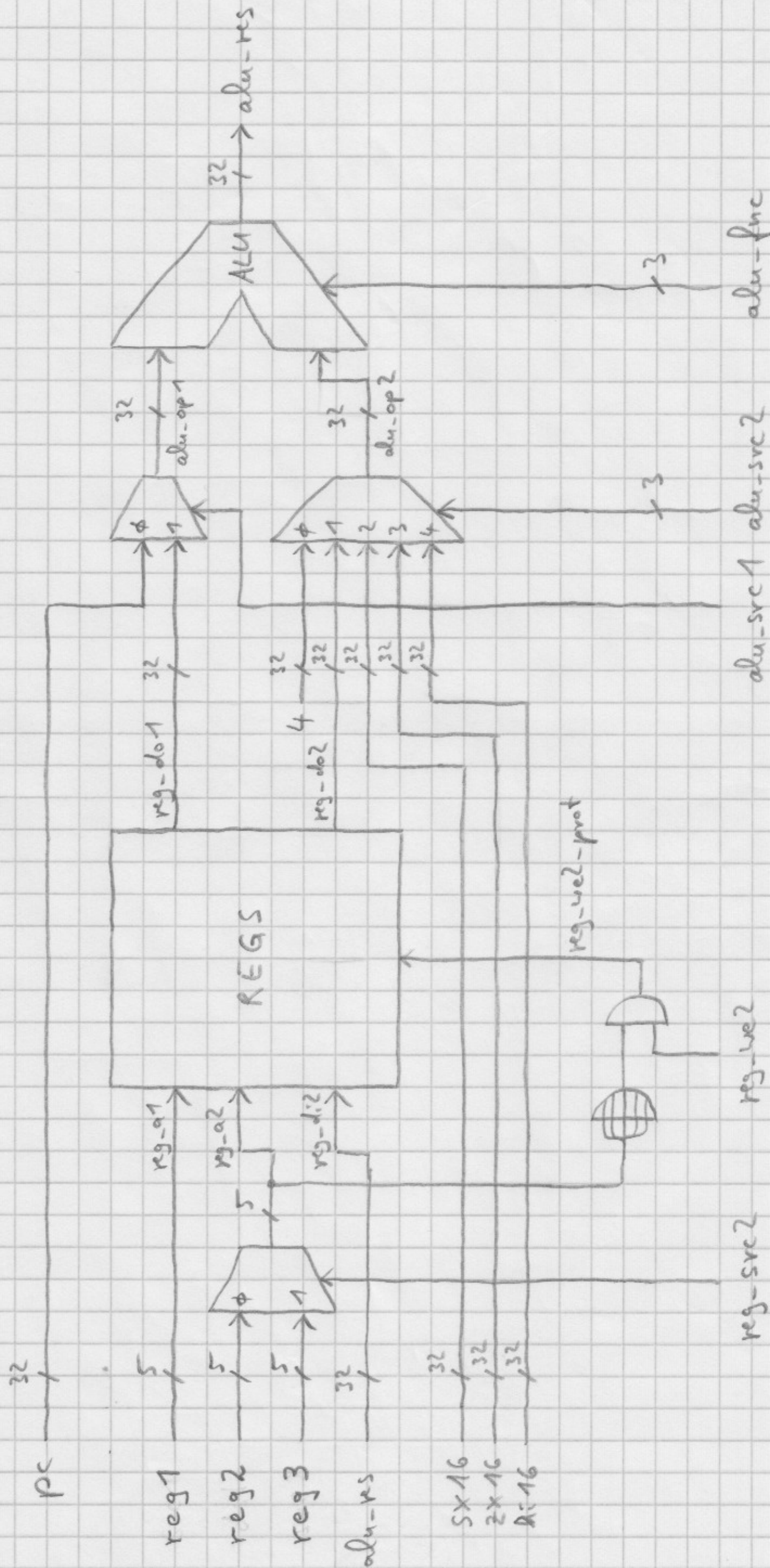


sx16 = sign-extend (16-bit-immediate)

zx16 = zero-extend (16-bit-immediate)

hi16 = { 16-bit-immediate, 16'h0000 }

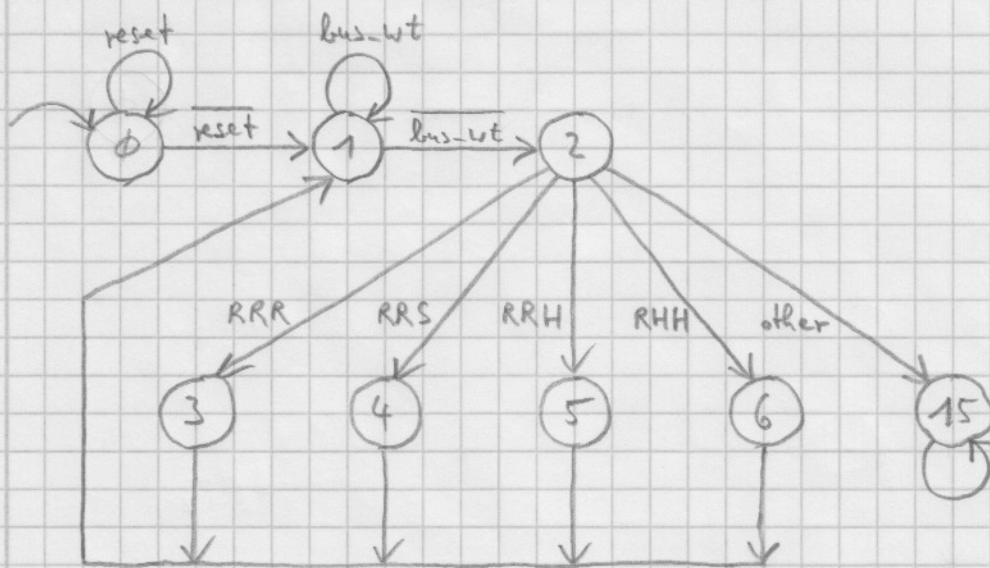
2.3.2.4. Datenpfad für Register- und Immediate-Befehle



2.3.2.5. Steuerwerk für Register- und Immediate-Befehle

Synchroner Automat mit den Zuständen:

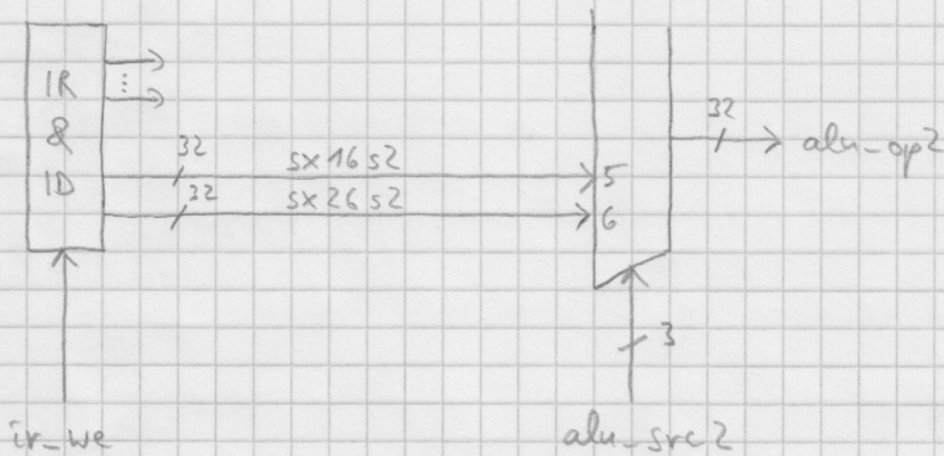
0	Reset
1	Fetch Instruction
2	Decode Instruction & Increment PC
3	Execute RRR-Instr (z.B. add $5, $6, $7)
4	Execute RRS-Instr (z.B. add $5, $6, 12)
5	Execute RRH-Instr (z.B. and $5, $6, 12)
6	Execute RHH-Instr (z.B. ldhi $5, 0x8000FFFF)
15	Error



Steuerwerk ist Moore-Automat: Kontrollsignale für den Datenpfad werden (nur) aus dem Zustand erzeugt. Dabei rechnet man formal das Instruktionsregister mit zum Zustand des Automaten.

2.3.2.6. Datenpfad für Sprünge

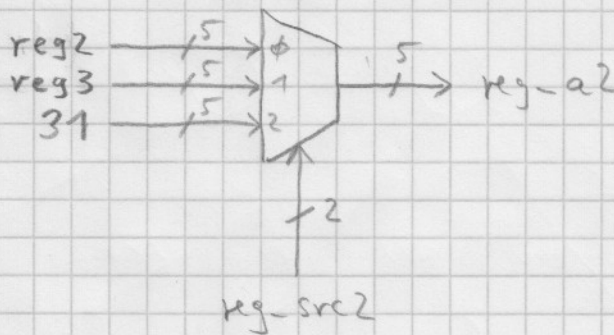
a) Ergänzungen am Instruktions-Dekoder / ALU-Operand 2-Selektor



sx16s2 = { sign-extend (16-bit-immediate), 2'b ϕϕ }

sx26s2 = { sign-extend (26-bit-immediate), 2'b ϕϕ }

b) Ergänzung zum Speichern der Return-Adresse



c) Berechnung der Entscheidungsgrundlage für bedingte Sprünge

Benötigte Information: $a = b$, $a <_u b$ ($<_u$: vorzeichenloser Vergleich)

Alle anderen Sprünge lassen sich daraus ableiten. (*)

α) $a = b \Leftrightarrow a - b = \phi$: ALU subtrahiert, dann NOR(result)

β) $a <_u b \Leftrightarrow a - b <_u \phi$ (???) ↯

Kleiner Trick: Fasse die vorzeichenlosen 32-Bit-Zahlen auf als positive vorzeichenbehaftete 33-Bit-Zahlen!

(*) Aber Achtung: $a > b \Leftrightarrow b < a \not\Leftrightarrow b \le a \Leftrightarrow \neg(b > a)$

$a' = \phi a_{31} \dots a_0 \quad , \quad b' = \phi b_{31} \dots b_0$

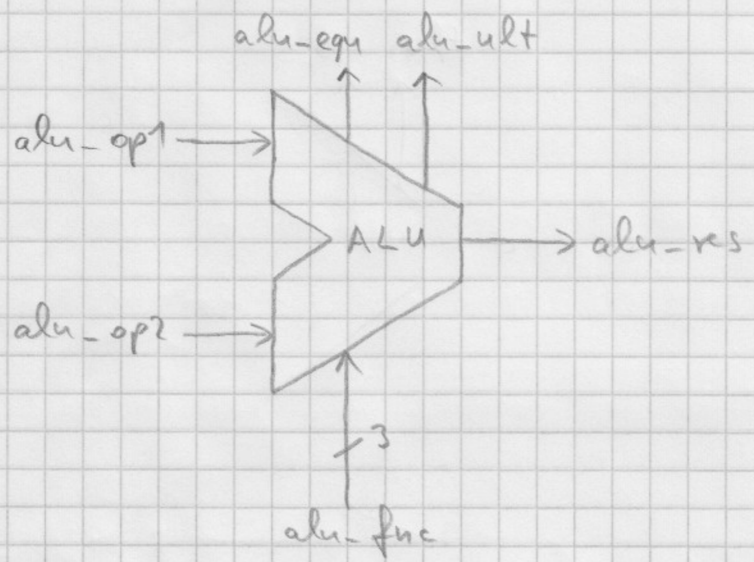
$a <_u b \iff a' < b' \iff a' - b' < \phi$

Die Differenz ist in 33 Bits darstellbar:

$0 \leq a \leq 2^{32} - 1 \quad , \quad 0 \leq b \leq 2^{32} - 1 \quad \Rightarrow$

$-(2^{32} - 1) \leq a - b \leq 2^{32} - 1$

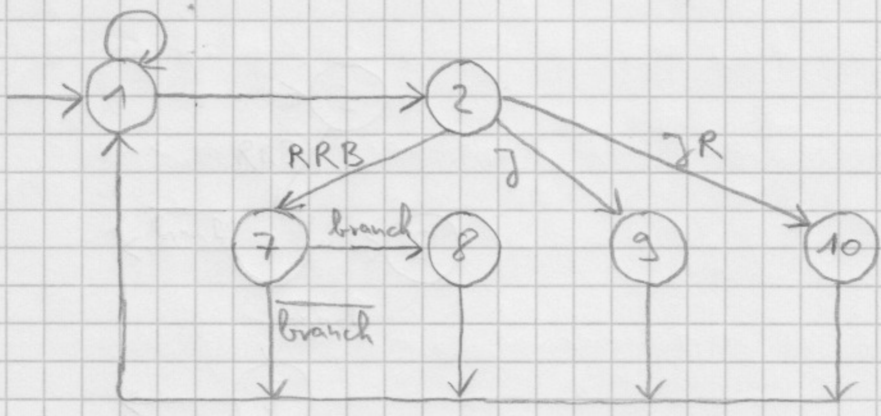
Also: $a <_u b \iff MSB(a' - b') = 1$



2.3.2.7. Steuerwerk für Sprünge

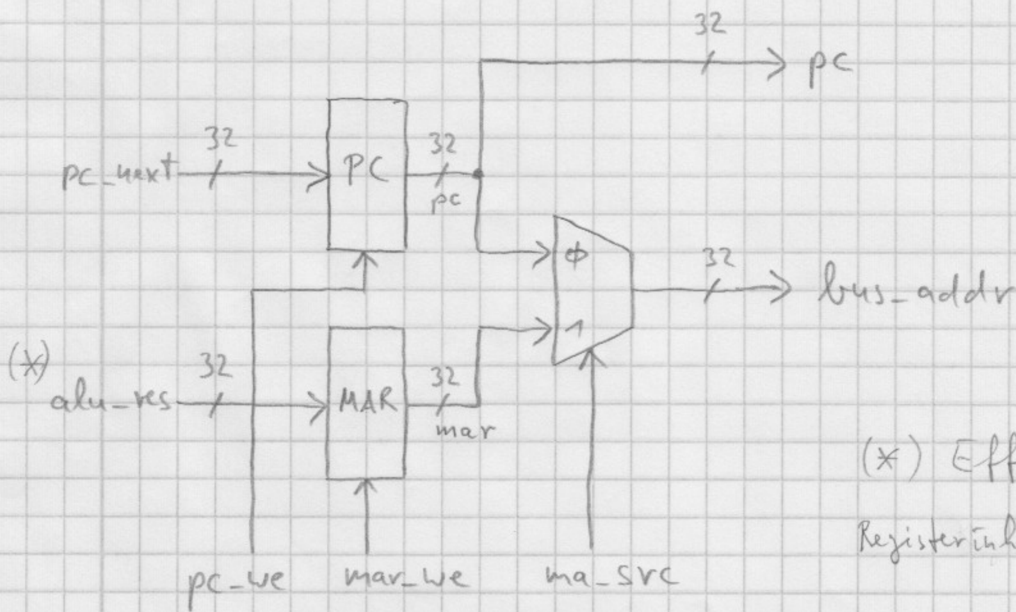
Zustände:

- 2 Speichert ggf. PC+4 in $31
- 7 Execute RRB-Instr (Subtraktion)
- 8 Execute RRB-Instr (neuer PC)
- 9 Execute J-Instr
- 10 Execute JR-Instr



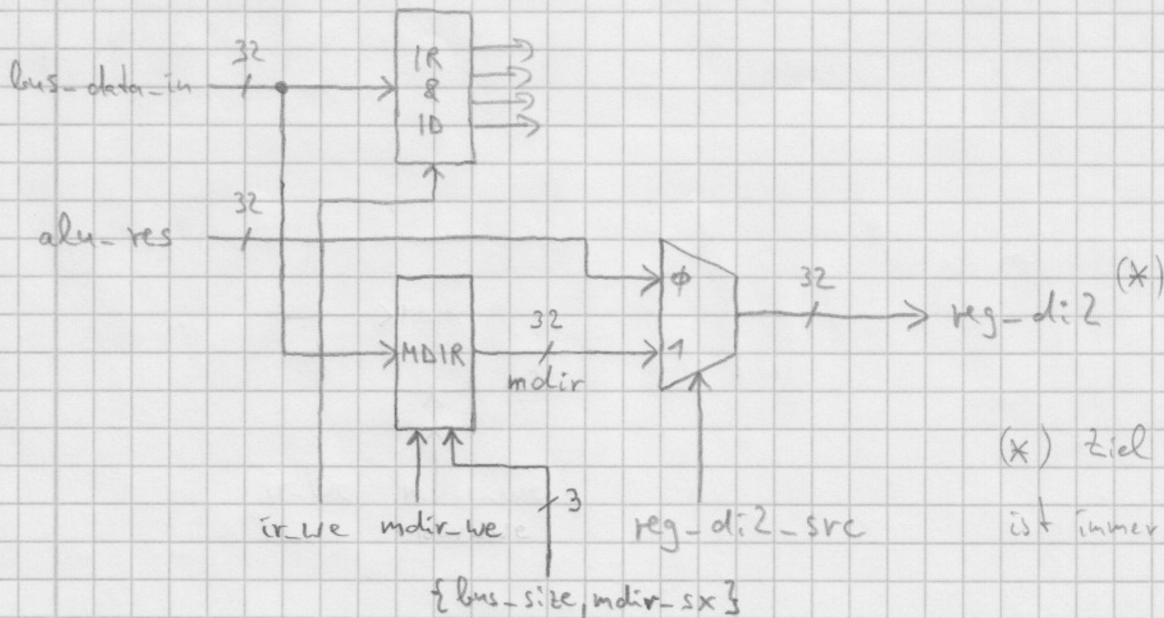
2.3.2.8. Datenpfad für Load/Store-Operationen

a) Bus-Adresse



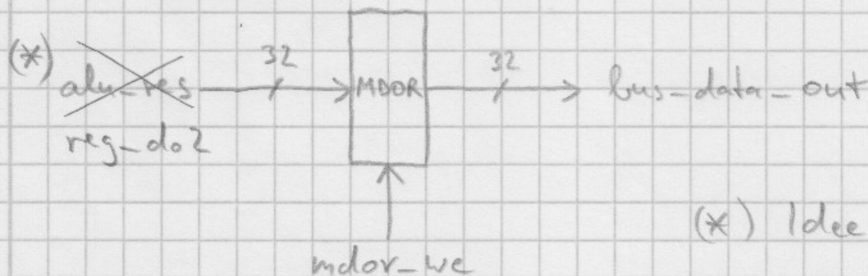
(*) Effektive Adresse = Registerinhalt + Immediate-Offset

b) Daten: Bus → CPU



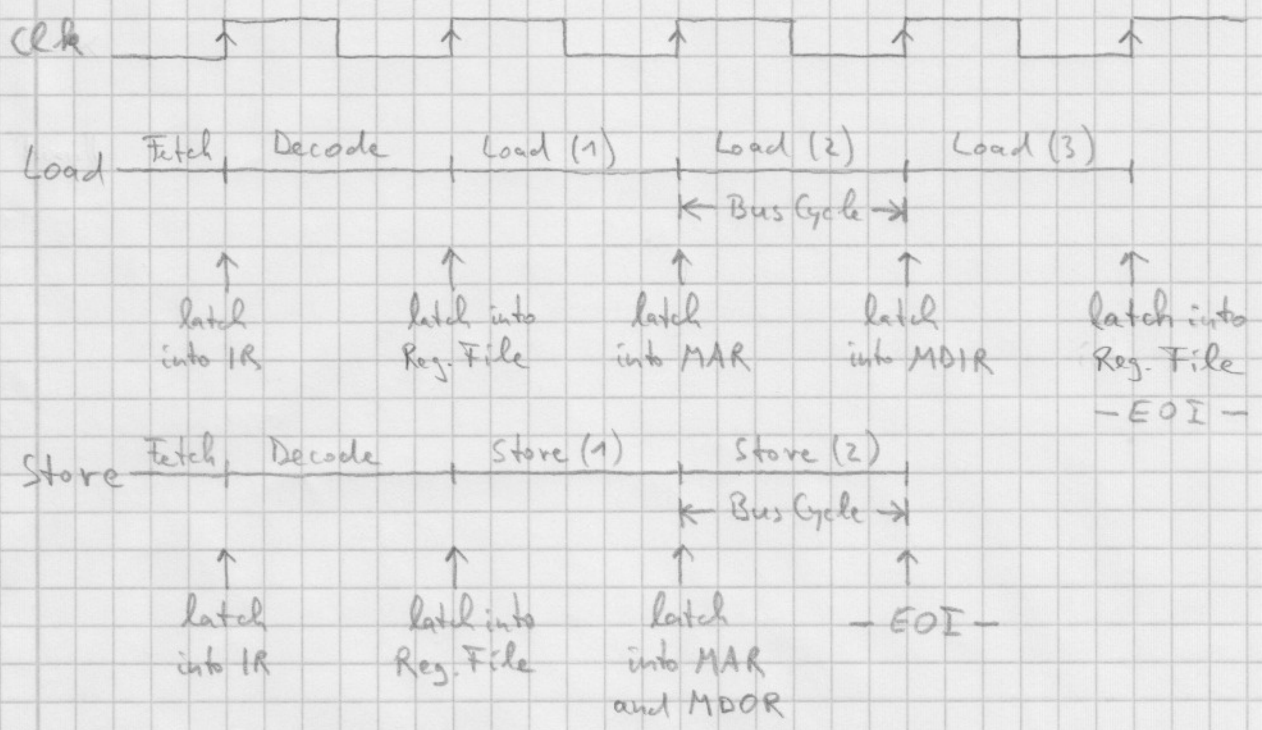
(*) Ziel des Transfers ist immer ein Register

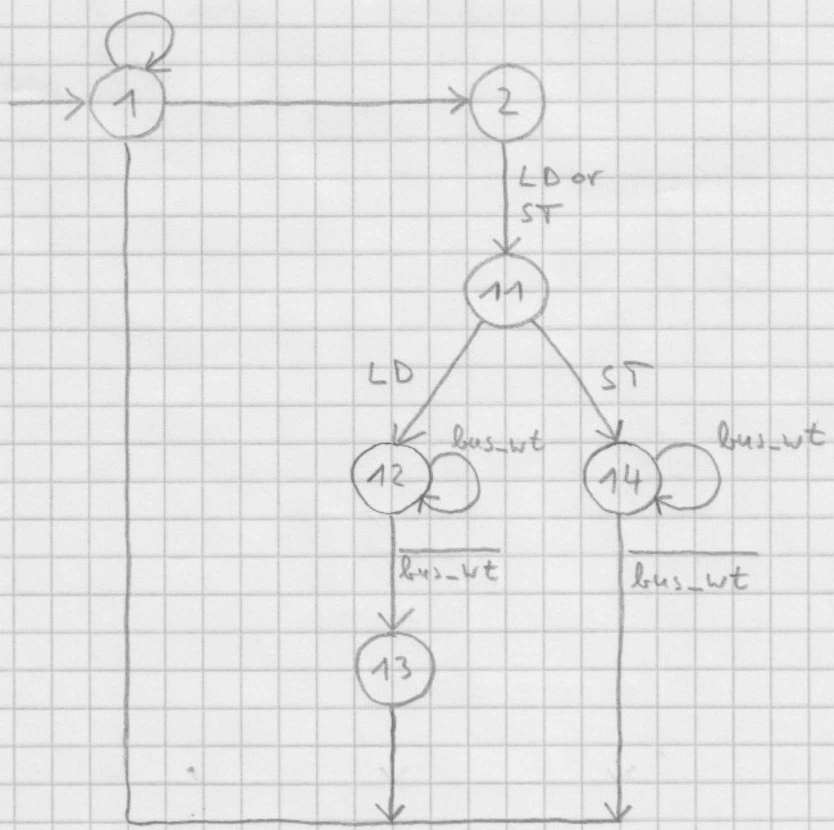
c) Daten: CPU → Bus



(*) Idee: Beide Pfade durch die Registerbank gleichzeitig nutzen!

2.3.2.9. Steuerwerk für Load/Store-Operationen





2.3.3. Interrupts und Exceptions

Interrupt: Unterbrechen des Kontrollflusses durch CPU-externes Ereignis; konzeptionell "zwischen" zwei Instruktionen (Bsp.: Plattenoperation beendet)

Exception: Unterbrechen des Kontrollflusses durch die momentan ausgeführte Instruktion (Bsp.: Division durch ϕ, illegaler Opcode, Zugriff auf geschützte Seite, TLB-Miss, etc). Achtung: die Instruktion darf nicht weiter ausgeführt werden!

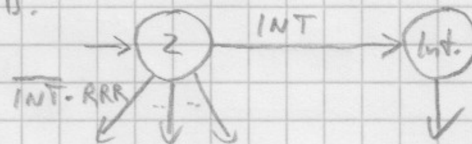
Software zur Behandlung von Interrupts und Exceptions benötigt mindestens:

- die Adresse der betroffenen Instruktion
- die Ursache der Unterbrechung

-> 2 Spezial-Register, die von Software gelesen werden können

Handhabung der Unterbrechung in der Multi-Cycle-Impl.:

- Zykl. Abfragen des Interrupt-Eingangs im Steuerwerk, z.B.



- Ergänzung des Steuerwerks um Zustände / Übergänge für Exceptions, z.B.: