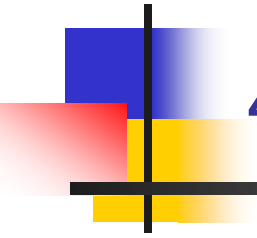


4. Steuerwerk und Operationswerk

Technische Grundlagen der Informatik 2
(Rechnertechnologie 2)
SS 2006

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen

Auf Basis von Material von

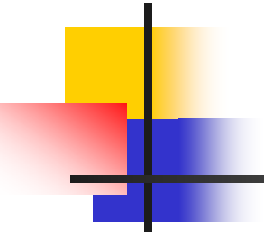
Rolf Hoffmann

FG Rechnerarchitektur

Technische Universität Darmstadt

Organisatorisches

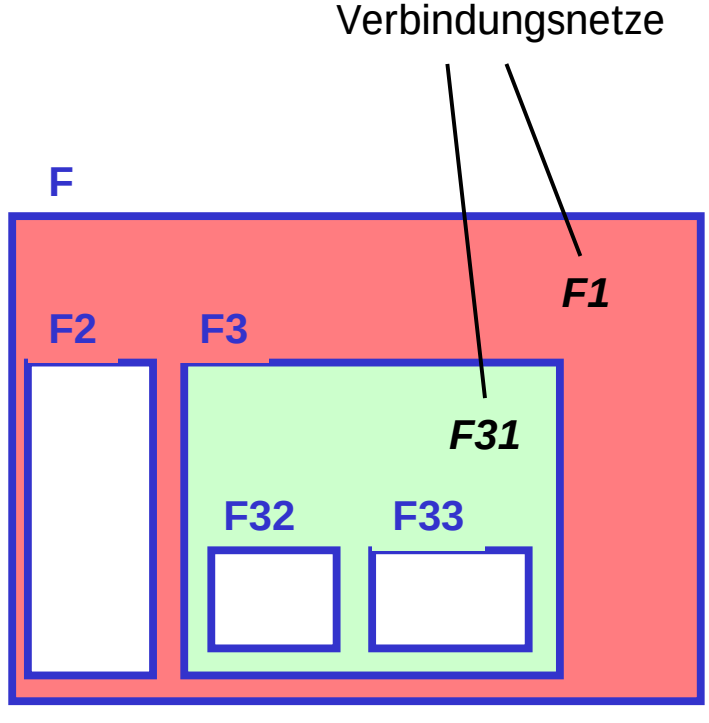
- Ausweichtermine für Brückentage
  - Freitags, 7.7. und 14.7., 14:25-16:05, S2|02 C205
- Informatik B: Anmeldung zur Semestralklausur (Scheinklausur) am 6.6.
  - Sekretariat des FG ESA, 12:00-16:00 Uhr, S2|02 E104
  - Aus Raumgründen unbedingt erforderlich!



Inhalt

4.1 Hierarchische Gliederung von Systemen

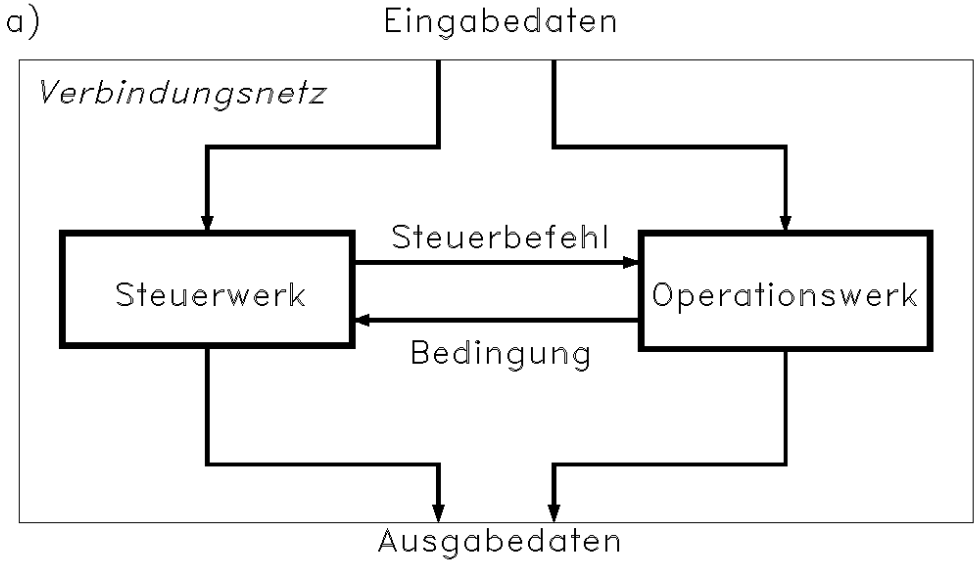
- **Hierarchische Gliederung**
  (Zerlegung, Zusammensetzung)
  - $F = F_1\{F_2, F_3, \ldots F_n\}$
  - $F_i = F_{i,1}\{F_{i,2}, F_{i,3}, \ldots F_{i,n(i)}\}$
  - Komponente = Komposition{Teilkomponenten}
  - Komponente := Komponente{Komponenten}
- **Funktionseinheit**
  - Funktionseinheit = Zusammenspiel{Funktionseinheiten}
  - Funktionseinheit = Verbindungsnetz{Funktionseinheiten}



4.2 Steueroperationssystem

- Funktionseinheit oft zusammengesetzt aus
  - **Steuerwerk** (Control Unit)
  - **Operationswerk** (Data Path)
- Zusammen bezeichnet als ***Steueroperationssystem***:

- ***Steueroperationssystem***
  = Verbindungsnetz{**Steuerwerk**, **Operationswerk**}



Steueroperationssystem(2)

- **Steuerwerk**
  - Sendet Steuerbefehle (Steuersignale) an das Operationswerk
  - Trifft Entscheidungen aufgrund der Bedingungen und Eingabedaten
  - Enthält ein Steuerprogramm (Steueralgorithmus, Mikroalgorithmus) zur
    - Analyse der Bedingungen
    - Synthese der Steuerbefehle
  - [Bedingung ->] Programmzustandsänderung -> Steuerbefehl
  - Aktiver Teil
- **Operationswerk**
  - Führt Operationen auf den Daten (Eingabedaten, gespeicherte Daten) aus
  - Meldet Bedingungen (Meldesignale, Statussignale) zurück an das Steuerwerk
  - Steuerbefehl -> Datenzustandsänderung [-> Bedingung]
  - Passiver Teil

4.3 Mikrooperationen

- **Synchrone Mikrooperation**

  si <- fi**(s,x)**

  Eine Operation fi auf einem synchronen Register si. Eingangswerte in die Funktion können sein: Bedingungen **x** und Werte **s** von
  - dem eigenen Register
  - anderen Quellen (Register, Signale von Wires, Werte von einfachen Speicherzellen)

- **Asynchrone Mikrooperation**

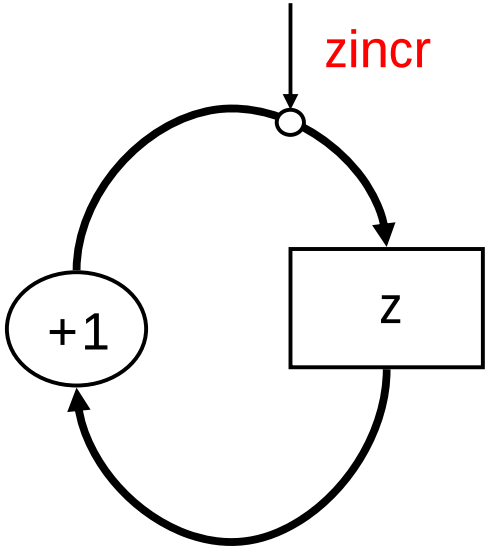
  si := fi**(s,x)**

  Eine Operation fi auf einer einfachen Speicherzelle si. Eingangswerte in die Funktion können Bedingungen **x** sein und Werte **s** von
  - andere Quellen (Register, Signale auf Wires, Werte von einfachen Speicherzellen)
  - verboten ist die Rückkopplung auf die eigene Speicherzelle (z.B. s:=s+1) weil sonst eine asynchrone Rückkopplung entsteht.

Parallele Ausführbarkeit

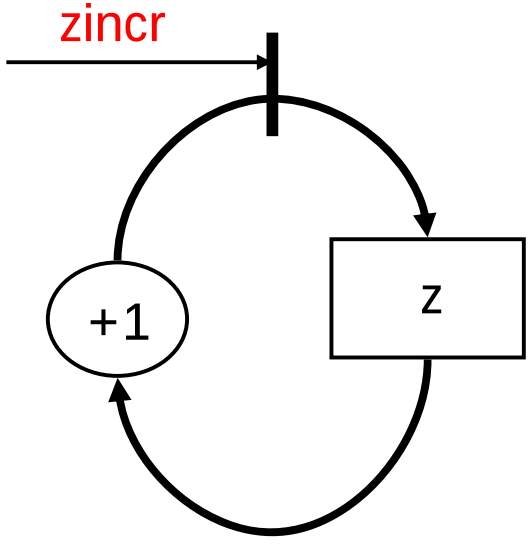
- **Synchrone µops** können synchron-parallel ausgeführt werden. Warum?
  - Synchrone Register bestehen aus einem Master- und einem Slave-Speicher
  - **Phase 1**: Mit den alten Werten, die in den Slaves (Registerausgänge) gespeichert sind, werden die Berechnungen durchgeführt. Am Ende dieser Phase werden die neu berechneten Werte in den Mastern der Register zwischengespeichert.
  - **Phase 2**: Die in den Mastern gespeicherten neuen Werte werden in die Slaves kopiert.
  - Die Phase 1 und die Phase 2 sind zeitlich entkoppelt.
  - Möglichkeiten zur Erzeugung der Phasen
    - **Zwei sich nicht überlappende Takte**
    - **Einflankengetriggert**: mit der positiven Taktflanke werden nacheinander die beiden Phasen angestoßen
    - **Zweiflankengetriggert**: Die Übernahme in die Slaves erfolgt mit der einen Taktflanke, das Kopieren vom Master in den Slave mit der anderen.
- **Asynchrone µops** können nur dann parallel ausgeführt werden, wenn keine asynchronen Rückkopplungen entstehen.

Synchrone Mikrooperation, Darstellung

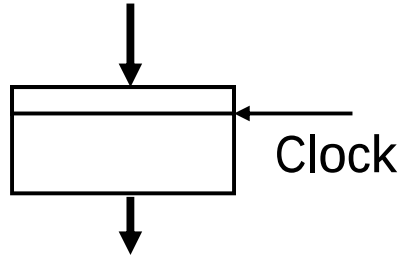


zincr: (z<=z+1)

Durch das Steuersignal zincr wird die synchrone Mikrooperation z<=z+1 aktiviert.

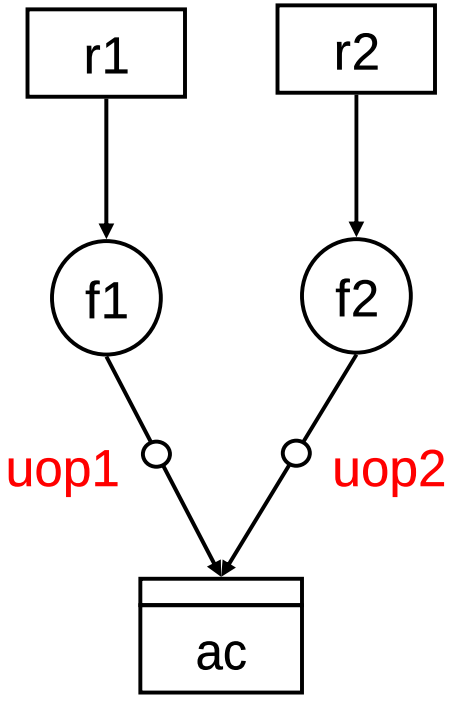


Alternative Darstellung



Diese Darstellung eines Registers verdeutlicht die synchrone Arbeitsweise

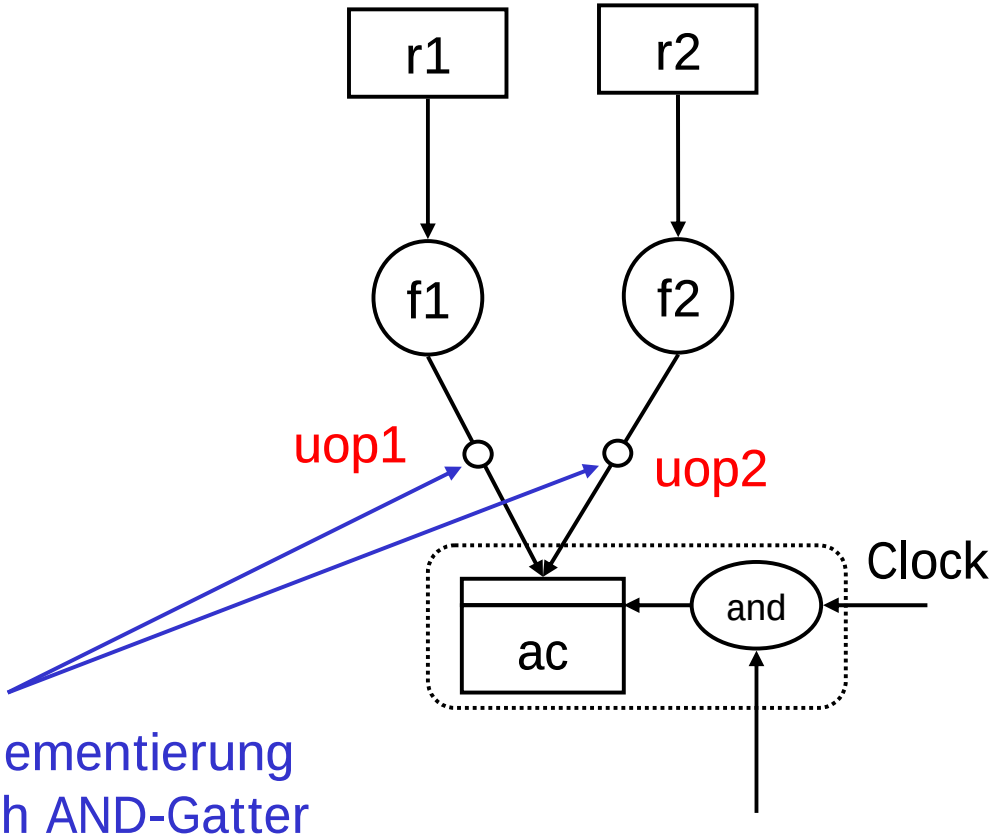
Zwei alternative Mikrooperationen



uop1: (ac<=f1(r1))

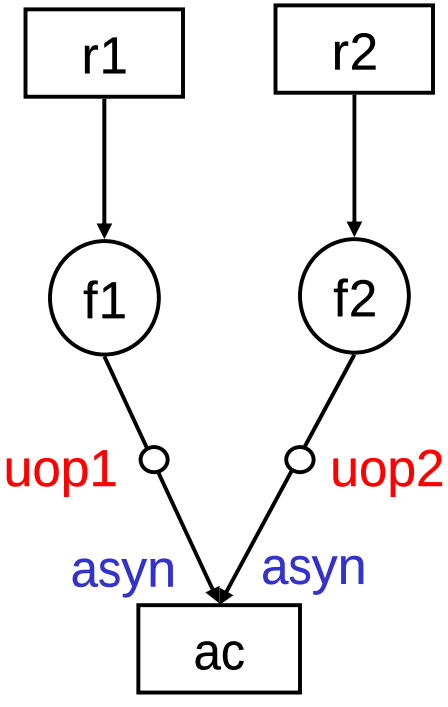
uop2: (ac<=f2(r2))

Implementierung der
Aktivierung durch ClockEnable



Implementierung
durch AND-Gatter
oder Multiplexer

Asynchrone Mikrooperation, Darstellung



uop1: (ac:=f1(r1))

uop2: (ac:=f2(r2))

Die asychrone Arbeitsweise wird durch den Zusatz „asyn“ zum Ausdruck gebracht.

Schreibweise der Zuweisung **:=**

In Verilog

```
reg ac, r1, r2;
if (uop1) ac=f1(r1);
if (uop2) ac=f2(r2);
```

4.4 Verschiedene Realisierungen von Steuerwerken

- Schaltwerk/Automat
  - s' = f(x,s) Zustandsfunktion
  - y = g(x,s) Ausgangsfunktion
  - s(t+1) = s'(t) Zeitverhalten

  - t = Folge von Zeitintervallen, in denen der Zustand stabil ist
  - s = Zustand, Adresse
  - s' = Folgezustand, Folgeadresse
  - x = Eingangssignale, Bedingung
  - y = Ausgangssignale, Steuerbefehl
- Realisierung von f und g
  - Hardware-Steuerwerk mit
    - Gatternetzwerk, Schaltmatrix (PLA), Platzbedarf minimiert
  - Mikroprogramm-Steuerwerk mit
    - Festwertspeicher, Mikroprogrammspeicher

4.4.1 Mikroprogramm-Steuerwerk

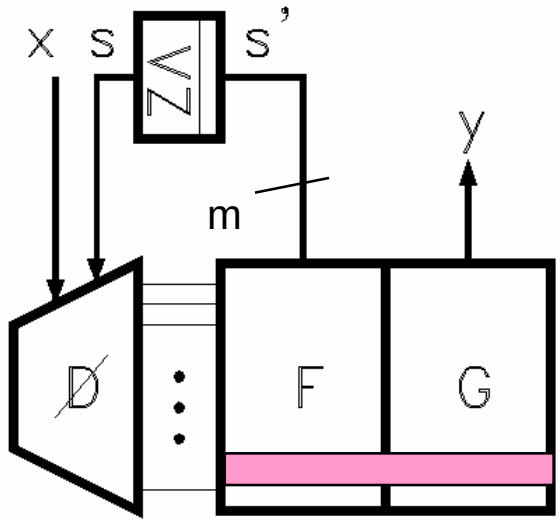
a)

Übergangstabelle

x_s	s'	y
D	F	G

b)

Mikroprogramm-Steuerwerk



[s] s' == F(x_s) , Y == G(x_s)

Ausgangspunkt ist die Übergangstabelle. D ist die Decodiermatrix, meist in dual aufsteigend sortierter Form Ø (alle möglichen Adressen von oo..o bis ll .. l).

Die Adresse ist die Konkatenation von der Bedingung x mit dem Zustand s.

Die ausgewählte Zeile von F gibt den aktuellen Folgezustand (Folgeadresse) an.

Die ausgewählte Zeile von G gibt die aktuellen Ausgangssignale (Steuerbefehl) an.

Die aktuelle Folgeadresse und der aktuelle Steuerbefehl bilden den aktuellen Mikrobefehl.

Beispiel

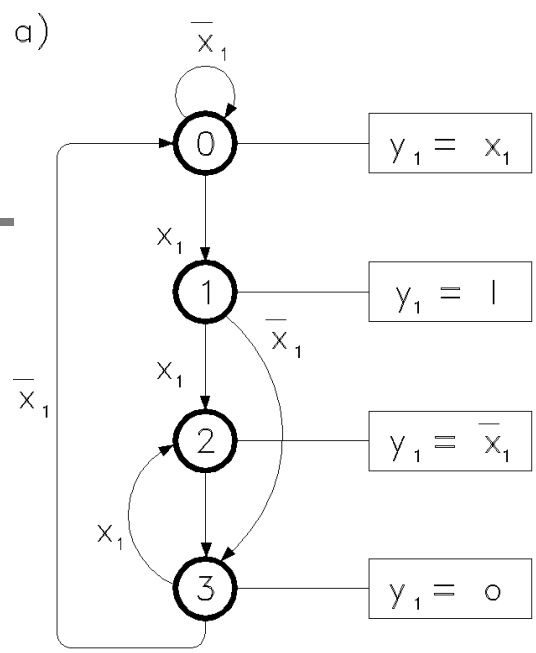
Zustandsdiagramm (a)

Mikroprogramm (b)

Übergangstabelle (c)

Realisierung (d)

a)



b)

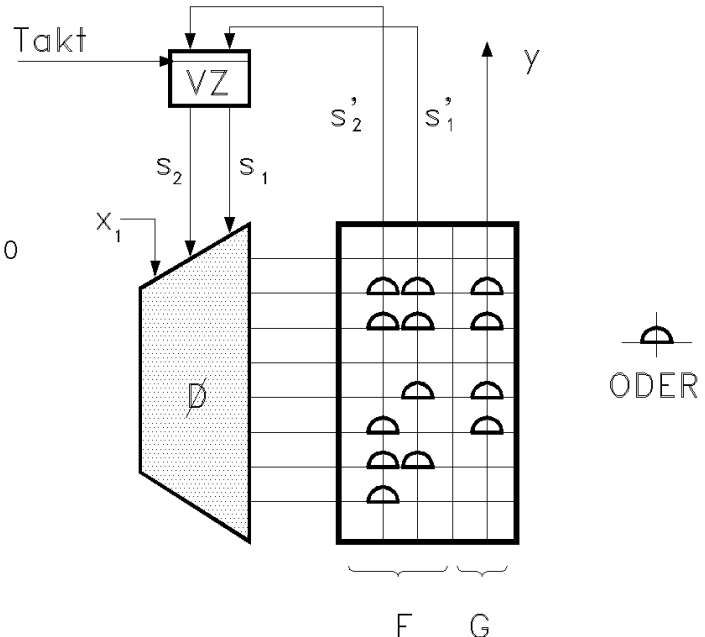
```
[0]  y1==x1, if x̄1 then next 0
                    else next 1 fi

[1]  y1==1  , if x̄1 then next 3
                    else next 2 fi

[2]  y1==x̄1, next 3

[3]  y1==0  , if x̄1 then next 0
                    else next 2 fi
```

c)

x_s	s'	y
$[x_1\ s_2\ s_1]$	$[s'_2\ s'_1]$	$[y_1]$
0 0 0	0 0	0
0 0 1	1 1	1
0 1 0	1 1	1
0 1 1	0 0	0
1 0 0	0 1	1
1 0 1	1 0	1
1 1 0	1 1	0
1 1 1	1 0	0
D	F	G

Mikro-befehl 0

d)

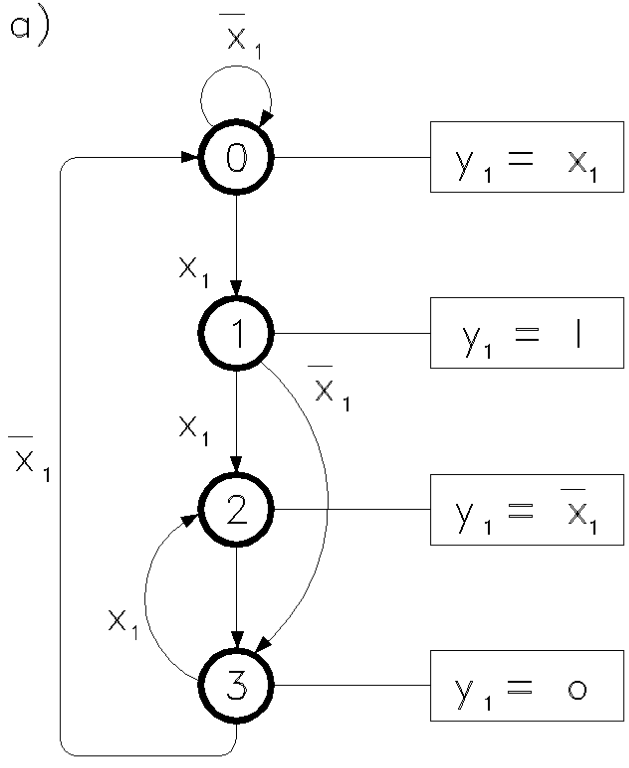


MP-Steuerwerk in Verilog


```
module stw(clk,x1,y1);
input  clk, x1;
output y1;
reg [1:0] s; // Zustandsregister sync
initial s=0:

reg [2:0] FG [0:7]; // ROM

wire [2:0] bef;
assign bef = FG[{x1,s}];
assign y1 = bef[0];

always @(posedge clk) s <= bef[2:1];
endmodule
```

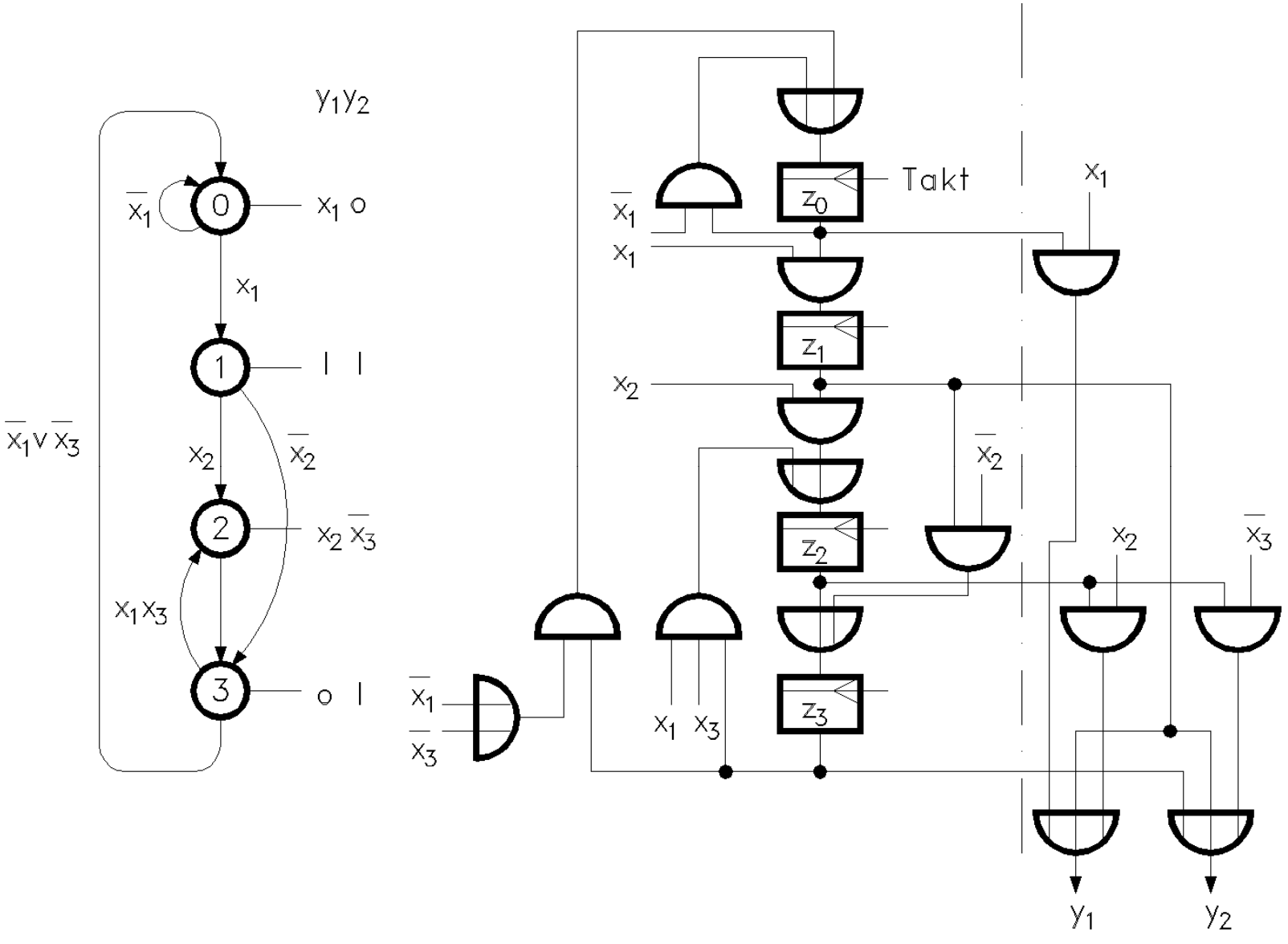
```
initial
begin
FG[0]=3‘b000;
FG[1]=3‘b111;
FG[2]=3‘b111;
FG[3]=3‘b000;
FG[4]=3‘b011;
FG[5]=3‘b101;
FG[6]=3‘b110;
FG[7]=3‘b100;
end
```

Beschreibung als Zustandsautomat in Verilog

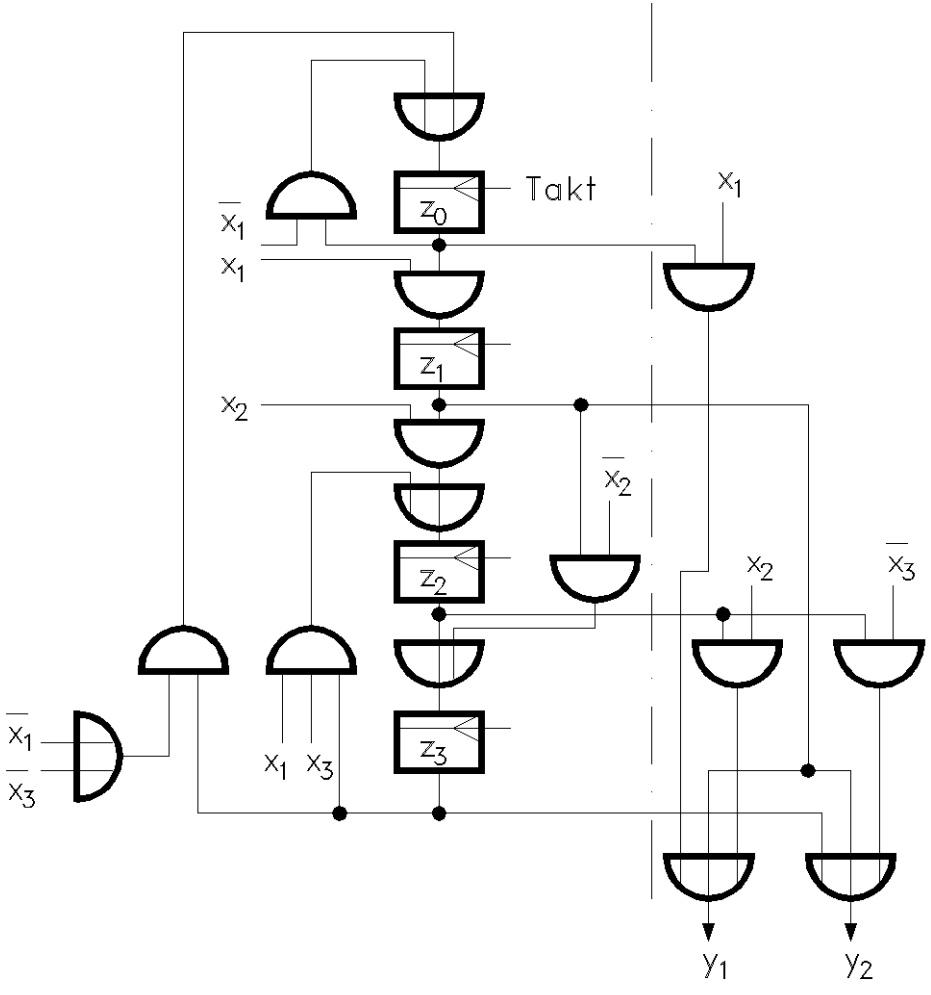



```
module stw(clk,x1,y1);
input  clk, x1;
output y1;
reg [1:0] z; // Zustandsregister sync
initial z=2'b00;
assign y1 = (z==0)&x1 | (z==1) | (z==2)&~x1;
always @(posedge clk)
   case(z)
   0: if(x1) z<=1; else z<=0;
   1: if(x1) z<=2; else z<=3;
   2: z<=3;
   3: if(x1) z<=2; else z<=0;
   endcase
endmodule
```

4.4.2 Schritt-Steuerwerk



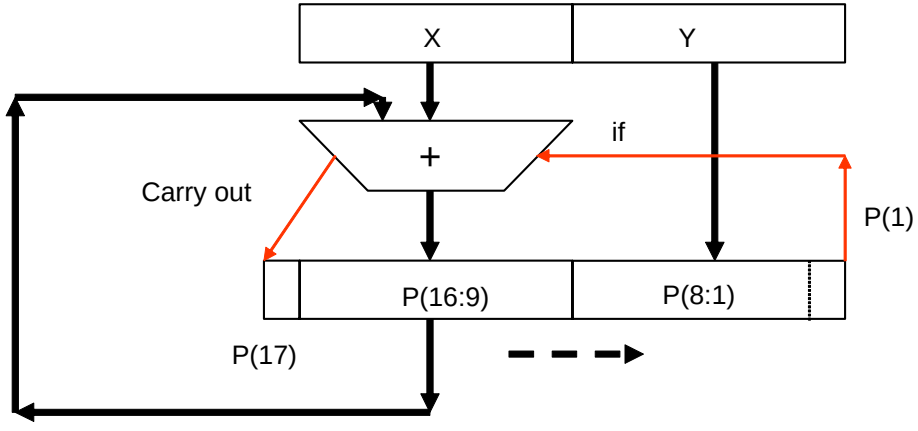
Schritt-Steuerwerk in Verilog




```
module stw(clk,x1,x2,x3,y1,y2);
input  clk,x1,x2,x3;
output y1,y2;
reg [3:0] z; // Zustandsregister sync
initial z = 4'b0001;
assign y1 = z[0]&x1 | z[1] | z[2]&x2;
assign y2 = ...
always @(posedge clk)
begin
  z[0] <= z[0]&~x1 | z[3]&(~x1|~x3);
  z[1] <= z[0]&x1;
...
end
endmodule
```

Fallbeispiel: Serien-parallele Multiplikation

4.5.1 **Algorithmische** Beschreibung




```
module mult2;
reg       [8:1]    x, y;
reg       [17:1]   p;
integer i;

initial
  begin
        x = 8; y = 255;
        p = {9'd0, y};
        for( i=0; i<=7; i=i+1 )
        begin
              if (p[1])  p[17:9] = p[16:9] + x;
              p = p >> 1;
        end
        $display("%d * %d = %d", x, y, p);
              // 8*255=2040
  end
endmodule
```

Beispiel: 8*255


```
vor             0: 0000000011111111
nach            0: 0000010001111111
vor             1: 0000010001111111
nach            1: 0000011000111111
vor             2: 0000011000111111
nach            2: 0000011100011111
vor             3: 0000011100011111
nach            3: 0000011110001111
vor             4: 0000011110001111
nach            4: 0000011111000111
vor             5: 0000011111000111
nach            5: 0000011111100011
vor             6: 0000011111100011
nach            6: 0000011111110001
vor             7: 0000011111110001
nach            7: 0000011111111000
```

4.5.2 Synchrones Mikroprogramm

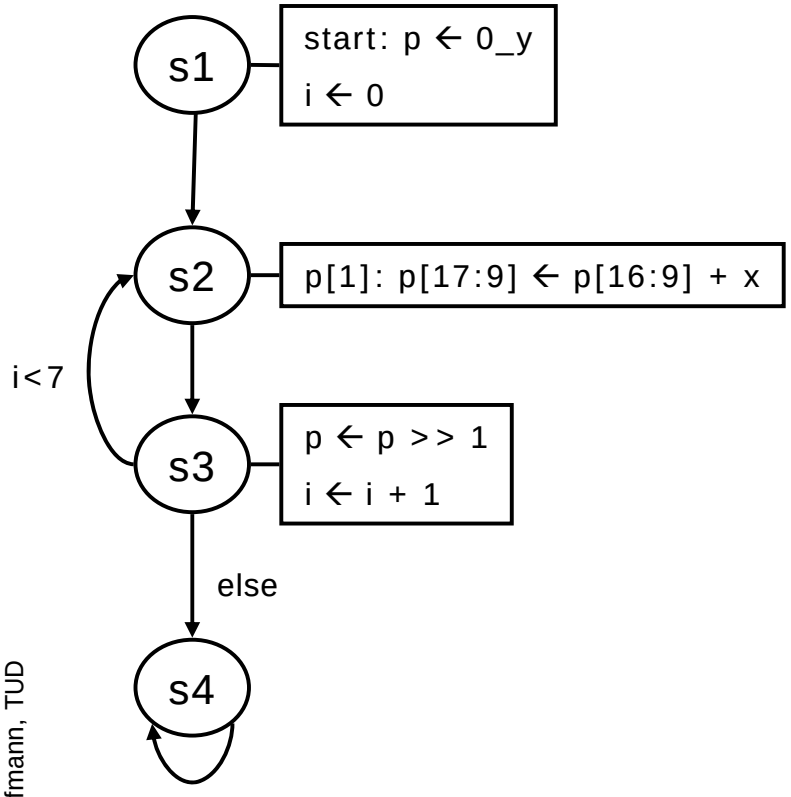
Enthält

* Ablauf (Zustandswechsel)
* die den Zuständen zugeordneten Mikrooperationen

- weitere Ausgangssignale.
- Aufbau ähnlich wie normales Programm
- eignet sich als Ausgangspunkt für die Hardware-Implementierung.
  - Mikrooperationen stehen direkt im Code.
  - Zerlegung in Steuerwerk und Operationswerk erfolgt später.

```
module mult3(clock, start, x, y, p, stop, state, i);
input    clock, start;
input    [8:1]   x, y;                    // von der Test-Umgebung
output   [17:1]  p; reg [17:1]  p;        // sync
output stop;
output   [1:0] state ; reg  [1:0] state ;   //sync
output   [2:0] i;  reg  [2:0] i;          // sync counter
parameter s1 = 'b00, s2 = 'b01,  s3 = 'b10,  s4 = 'b11;
initial state=s1;
```

Synchrones Mikroprogramm (2)

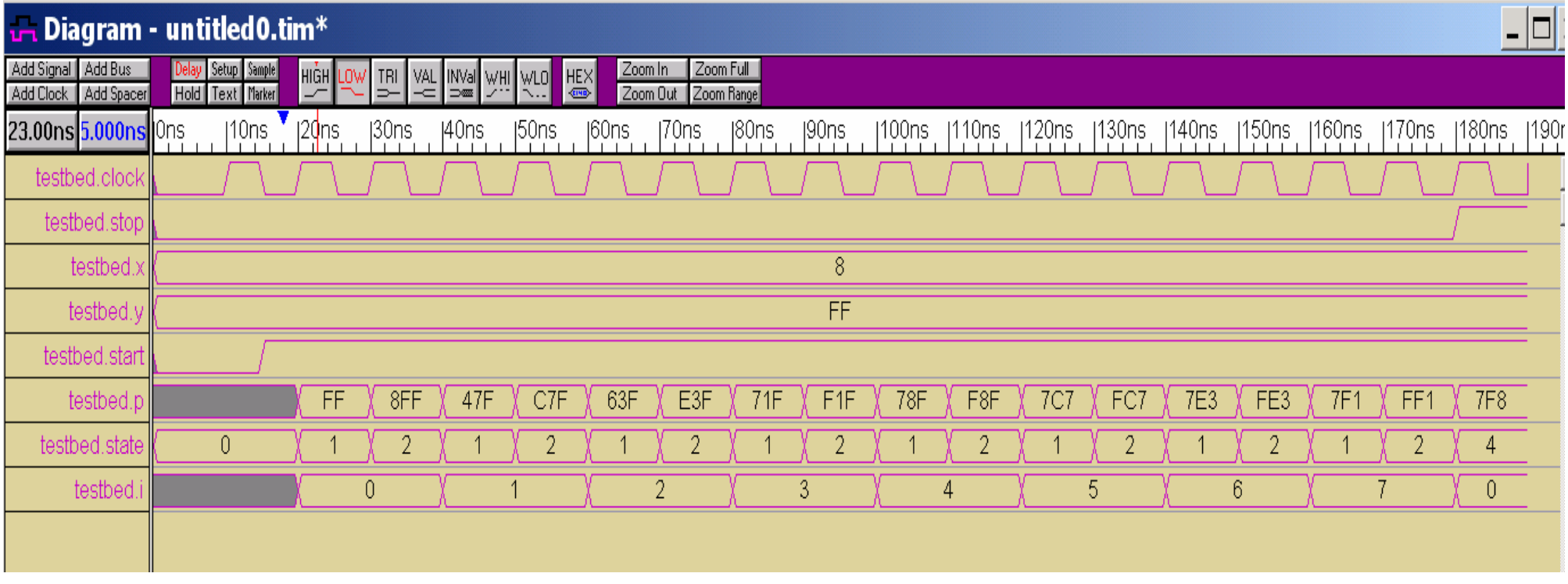



```
assign stop=(state==s4);
always @(posedge clock)
   begin
      case (state)
      s1: if (start) begin p <={9'b0, y}; i <= 0; state <= s2; end
      s2: begin
               if (p[1]) p[17:9] <= p[16:9] + x;
               state <= s3;
            end
      s3: begin
                  p <= p >> 1;
                  if ( i<7 ) state <= s2; else state <= s4;
                  i <= i + 1;
               end
      s4: state<=s4;
      default:   state <= s1;
      endcase
   end
endmodule
```

Testumgebung


```
module testbed();
reg clock;              // boole buffered
reg [8:1] x, y;         // boole, Eingangswerte
reg start;              // boole, start-signal
wire stop;              // stop signal vom Multiplizierer
wire [17:1] p;          // zur Anzeige des Ergebnisses
wire [1:0] state ;      // Anzeige
wire [2:0] i;           // Anzeige
mult3 multiplizierer(clock, start, x,y,p, stop, state, i);
initial begin x=8; y=255; end  // zu multiplizierende Werte
initial begin start=0; #15 start=1; end
initial begin clock=0; #5 repeat (100) #5 clock = ~clock; end
initial
   begin
    wait (stop);
    $display("%d * %d = %d  %h", x, y, p, p);
    @(posedge clock) $finish;
    end
endmodule
```

Timing-Diagramm



4.5.3 Zerlegung in Steuerwerk und Rechenwerk

- **Aufgabenstellung**
  - Das synchrone Mikroprogramm ist in ein Steuerwerk und in ein Rechenwerk zu zerlegen.
  - Der Zähler i soll sich im Rechenwerk befinden
  - Eine geeignete Schnittstelle (Steuersignale und Meldesignale) ist zu definieren.

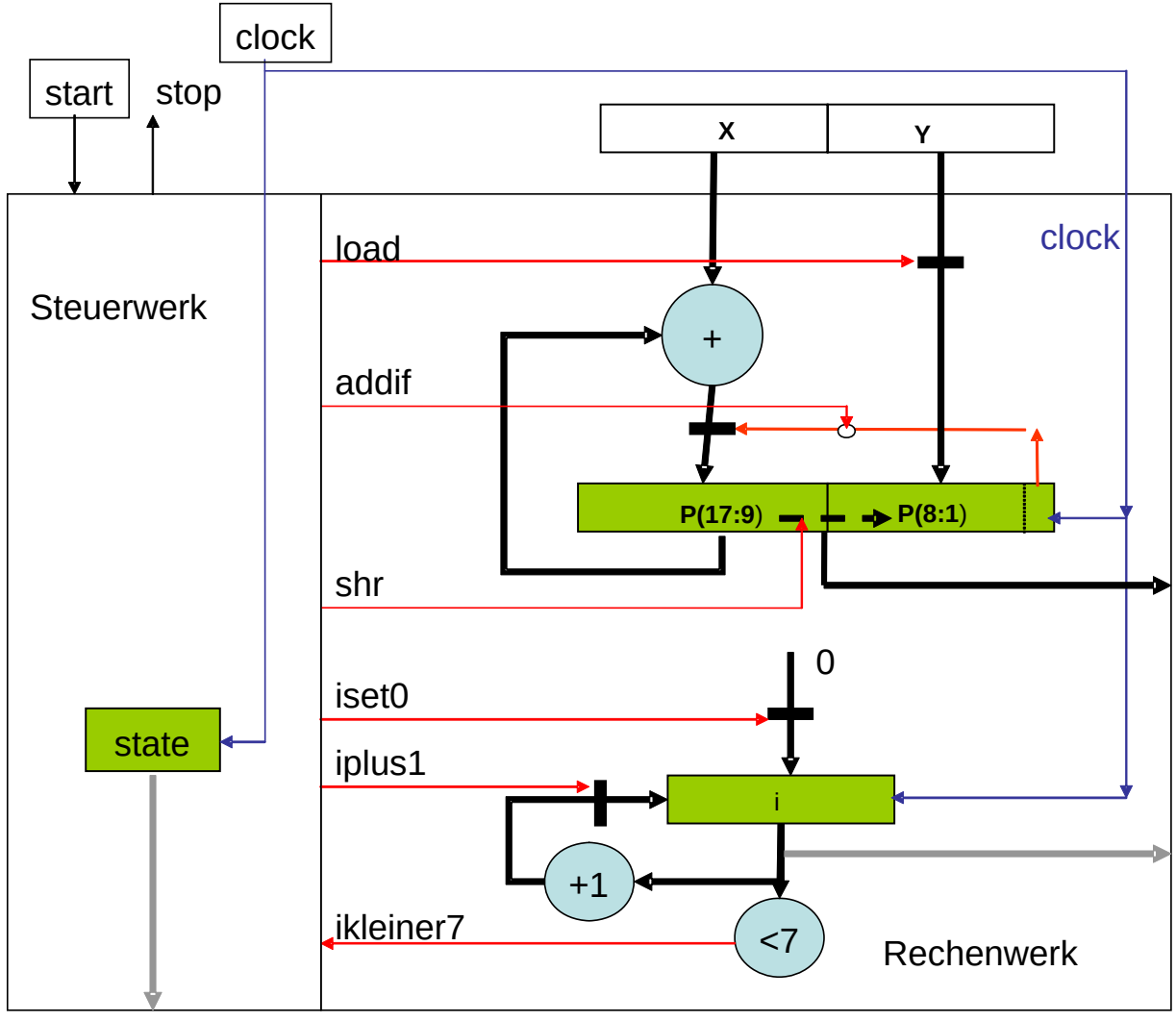
- **Vorgehensweise**
  - Ersetze die Mikrooperationen durch Steuersignale
    - z.B. load: (p <={9'b0, y)
      oder shr: (p <= p >> 1)
  - Definiere Meldesignale
    - hier: ikleiner7: (i <7)
  - Zeichne die Zerlegung (insbesondere das Rechenwerk) mit der Schnittstelle
  - Beschreibe das Steuerwerk und das Rechenwerk unter Verwendung der Schnittstelle

Schnittstelle

- **Mikrooperationen**
  - load: p <={9'b0, y}
  - addif: if (p[1]) p[17:9] <= p[16:9] + x
  - shr: p <= p >> 1
  - iset0: i <= 0
  - iplus1: i <= i+1

- **Meldesignale**
  - ikleiner7: ikleiner7=(i<7)

Steuerwerk-Schnittstelle-Rechenwerk



Steuerwerk (1):


```
module stw(ikleiner7, clock, start, stop, state, load, addif, shr, iset0, iplus1);
input    clock;
output   stop;                          // zur Testumgebung
output load, addif, shr;                // Steuersignale für p
output iset0, iplus1;                   // Steuersignale für i

input    start;                         // von der Testumgebung
input    ikleiner7;                     // Meldesignal

parameter s1 = 'b00, s2 = 'b01,  s3 = 'b10,  s4 = 'b10;
output   [3:0] state ; reg   [3:0] state ; initial state=s1;
```

Steuerwerk (2): Zustandsübergänge


```
always @(posedge clock)
    begin
        case (state)
        s1: if (start) state <= s2;                          // load, iset0 erzeugen
        s2: begin state <= s3; end                           // addif
        s3: if (ikleiner7) state <= s2; else state <= s4;    // shr, iplus1
        s4: state<=s4;
        endcase
    end
```

Die Ausgangssignale werden separat beschrieben (s. nächste Folie)

Steuerwerk (2): Ausgangssignale


```
assign  stop  = (state==s4);

assign load  = (state==s1)&start;
assign addif = (state==s2);
assign shr   = (state==s3);

assign iset0  = (state==s1);
assign iplus1 = (state==s3);
```

Rechenwerk

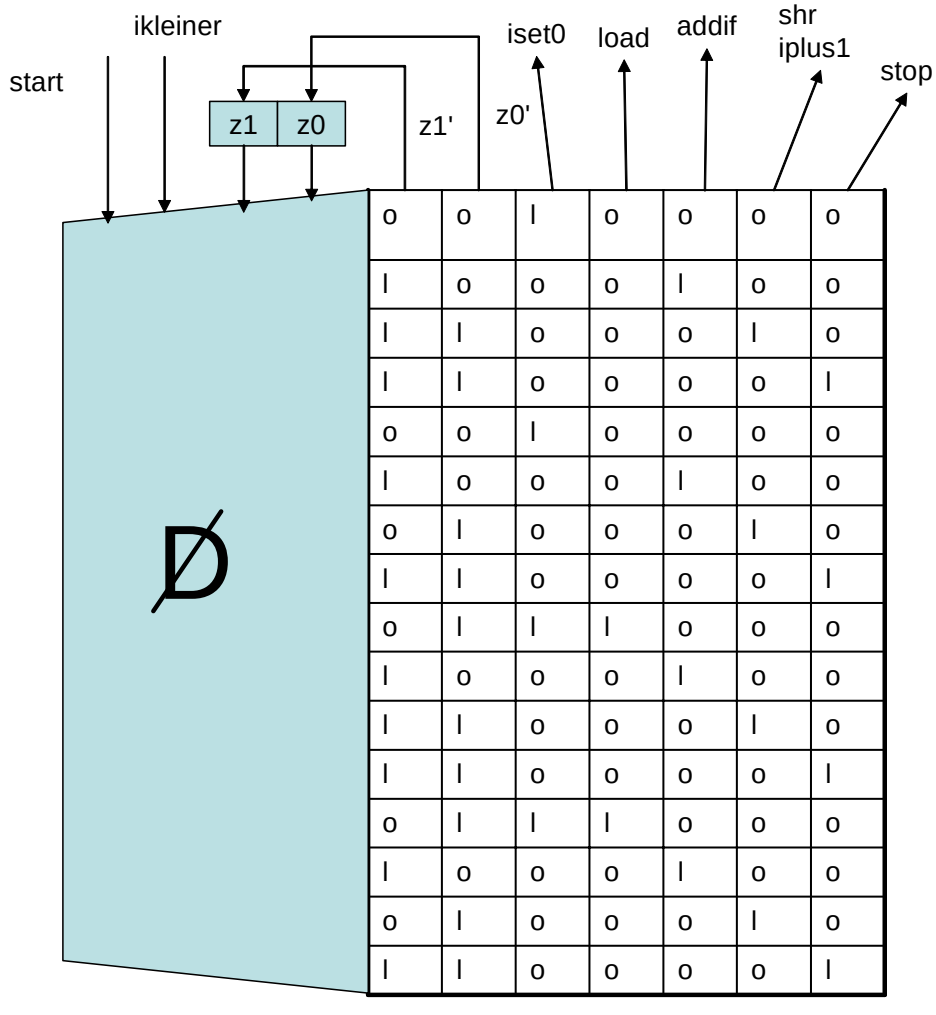

```
// rw enthält zähler i
module rw(iset0, iplus1,
      load, addif, shr, clock, x,y,p, i, ikleiner7);
input   clock;
input   [8:1]   x, y;  // von der Test-Umgebung
input   iset0, iplus1;
input   load, addif, shr;
output  [17:1]  p; reg [17:1]  p;   // sync
initial p=0:
output  [2:0] i; reg  [2:0] i;    // sync counter
initial i=0;

output ikleiner7;
assign ikleiner7=(i<7);


always @(posedge clock)
   begin
      if (load) p <={9'b0, y};
      if (addif) if (p[1]) p[17:9] <= p[16:9] + x;
      if (shr)  p <= p >> 1;
   end
always @(posedge clock)
   begin
      if (iset0)  i <= 0;
      if (iplus1) i <= i+1;
   end
endmodule
```

testbed


```
module testbed();
reg  clock;                    // boole buffered
reg  [8:1] x, y;               // boole, Eingangswerte
reg  start;                    // boole, start-signal
wire stop;                     // stop signal vom Multiplizierer
wire [17:1] p;                 // zur Anzeige des Ergebnisses
wire [1:0] state ;             // Anzeige
wire [2:0] i;                  // Anzeige
stw stw1(ikleiner7, clock, start, stop, state, load, addif, shr, iset0, iplus1);
rw  rw1(iset0, iplus1, load, addif, shr, clock, x, y, p, i, ikleiner7);
initial begin x=8; y=255; end  // zu multiplizierende Werte
initial begin start=0; #15 start=1; #10 start=0; end
initial begin clock=0; #5 repeat (100) #5 clock = ~clock; end
initial
    begin
    wait (stop);
    $display("%d * %d = %d  %h", x, y, p, p);
    @(posedge clock) $finish;
    end
endmodule
```

Mikroprogramm-Steuerwerk

- Vorgehensweise
  - Aufstellen einer dual aufsteigend sortierten Übergangstabelle.
  - Die Anzahl der Zeilen und der Zeilenindex ergibt sich durch die Konkatenation von x (Eingangssignale) und dem Zustand z.
  - Für jeden Zeilenindex sind der Folgezustand und die Ausgangssignale einzutragen.
  - Implementierung der Tabelle durch ein ROM und ein Zustandsregister
  - Verilog: analog zu Folie 14



o	o	l	o	o	o	o
l	o	o	o	l	o	o
l	l	o	o	o	l	o
l	l	o	o	o	o	l
o	o	l	o	o	o	o
l	o	o	o	l	o	o
o	l	o	o	o	l	o
l	l	o	o	o	o	l
o	l	l	l	o	o	o
l	o	o	o	l	o	o
l	l	o	o	o	l	o
l	l	o	o	o	o	l
o	l	l	l	o	o	o
l	o	o	o	l	o	o
o	l	o	o	o	l	o
l	l	o	o	o	o	l