



A. Koch

Eingebettete Prozessorarchitekturen

12. Programmierung Adaptiver Computer

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Wintersemester 2006/2007



A. Koch

Verilog



- Hardware-Beschreibungssprache
- Auszugsweise bereits behandelt in
 - Technische Grundlagen der Informatik II
 - Kanonik Computer Microsystems
- Erlaubt
 - Präzise Beschreibung von Hardware
 - Unterschiedlichste Berechnungsmodelle
 - Datenfluß
 - Endliche Automaten
 - Zeitliche Verteilung von Berechnungen
 - ...
 - Benutzung
 - **Simulation** ohne konkrete Hardware
 - **Synthese** von Hardware
- Hier nur **auszugsweise** Wiederholung einiger Sprachelemente

Einige Verilog-Sprachelemente



A. Koch

- Hierarchische Beschreibung
 - Aufteilen einer Schaltung in Unter**module**
 - Schnittstellen zwischen Modulen
- Parallele Abläufe
 - Reale Verarbeitung erfolgt i.d.R. **parallel**
 - Simuliert in **zufälliger** Reihenfolge
 - Jede Reihenfolge muß **explizit** formuliert werden
- Weitere Eigenschaften: Modellierung von
 - Zeitpunkten und -intervallen
 - Elektrischen Eigenschaften
 - ...

➡ Hier **nicht** behandelt

Verwende Verilog als **parallele Programmiersprache** für ACS.

Beispiel: Triviales Verilog-Modul



A. Koch

```
// Schnittstelle
module somelogic(x,y,a,b,c);
    input  [7:0] a, b, c;
    output [7:0] x, y;

    // Deklaration mit Zwischenberechnung
    wire  [7:0] temp = 8'd42 ^ c;

    assign y = (a & b) | temp;
    assign x = (a ^ b) | ~temp;
endmodule
```

- Modulkopf
- Schnittstellenbeschreibung
 - Angabe von Bit-Breiten
- **Parallele** Berechnungen
 - Im Simulator in
 - zufälliger Reihenfolge ausgeführt

Sequentielle Ausführung



A. Koch

- Zufällige Abarbeitungsreihenfolge störend
- Inseln pseudo-sequentieller Abarbeitung durch **Blöcke**

```
module somecalc(x,y,a,b,c);  
  input  [7:0] a, b, c;  
  output [7:0] x, y;  
  reg [7:0] x, y; // Variablen für Block  
  // wann soll Block ablaufen?  
  always @(a or b or c) begin  
    if (c) begin  
      x = a+10;  
      y = x+b;  
    end else begin  
      x = a-10;  
      y = x-b;  
    end  
  end  
endmodule
```

- **Sensitivity List** startet Blockausführung
 - Hier: **Änderungen** der Signale **a**, **b** oder **c**
- Zuweisungsziele in sequentiellem Block
 - Mit **reg** als **Register** deklarieren
- Anweisungen innerhalb von Block laufen **sequentiell** ab

Parallelität **innerhalb** von Blöcken



A. Koch

- Hardware rechnet parallel
- Daher in der Regel schneller als sequentielle Software
- Viele Feinheiten der Parallelität beschreibbar
- Nützlich auch **innerhalb** von Blöcken
 - **Getrenntes** Lesen und Schreiben von Variablen durch Zuweisung mit \leq (*non-blocking assignment*)
- Erst die **rechten** Seiten aller \leq lesen
 - In der Reihenfolge ihres Auftretens im Block
 - Damit weiterrechnen
- Erst am Ende des Blockes **linke** Seiten von \leq schreiben
 - Mit den im Laufe des Blockes berechneten Werten
- Effekt: Solche Zuweisungen laufen **parallel** ab

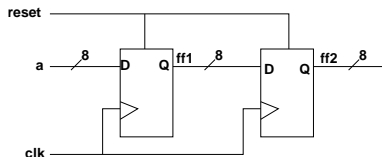
➡ Wofür gut? Kommt gleich ...

Getaktete Blöcke



A. Koch

- Alle Blöcke laufen **parallel** zueinander ab
 - In Hardware **echt** parallel
 - In der Simulation in **zufälliger** Reihenfolge
- Inter-Block Koordination mittels **Takt**



```
module ff2(ff2, a, clk, reset);
  input  [7:0] a;
  input      clk, reset;
  output [7:0] ff2;
  reg [7:0] ff1, ff2;
  always @(posedge clk) begin
    if (reset) begin
      ff1 <= 0; ff2 <= 0;
    end else begin
      ff1 <= a;
      ff2 <= ff1;
    end
  end
end
endmodule
```

- Beachte **parallele** Zuweisungen (<=)
 - Reset und eigentlicher Schiebевorgang



A. Koch

Kommunikation mit der RCU



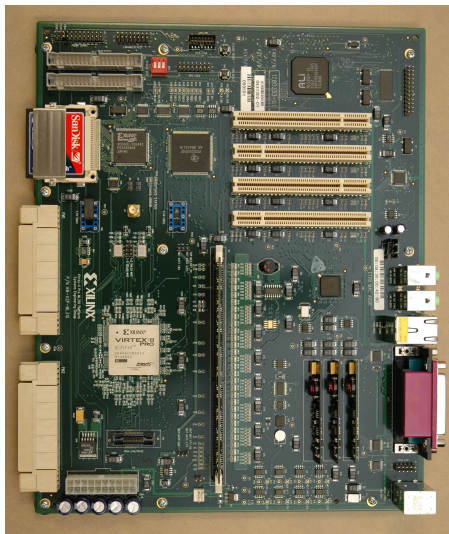
- Einfach bei
 - Rekonfigurierbare Funktionseinheit
 - Rekonfigurierbarem Koprozessor
- Hier in der Regel spezielle Prozessorbefehle vorhanden
 - Tensilica Xtensa LX, Stretch S5000
- Loser integrierte RCUs werden häufig in den Speicher eingeblendet
 - *memory mapped*
 - Lese und Schreibzugriffe der CPU betreffen nicht den Hauptspeicher
 - Sondern landen auf der RCU
 - **Akzeptieren** von Daten beim Schreiben
 - **Generieren** von Daten beim Lesen

Beispiel: ML310 Entwicklungsplattform

PC-artige Grundarchitektur



A. Koch



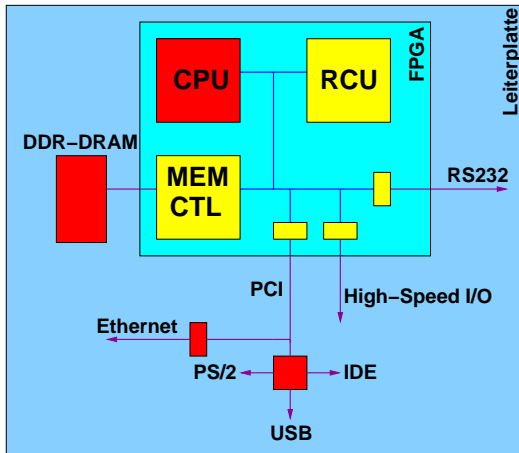
- Xilinx Virtex II Pro FPGA
- Zwei PowerPC-Prozessoren On-Chip
- 64MB DDR-DRAM
- Viele Schnittstellen
- Betrieb unter Linux

ML310 als ACS: Architektur

Feste Hardware in Rot, rekonfigurierbare Hardware in Gelb



A. Koch

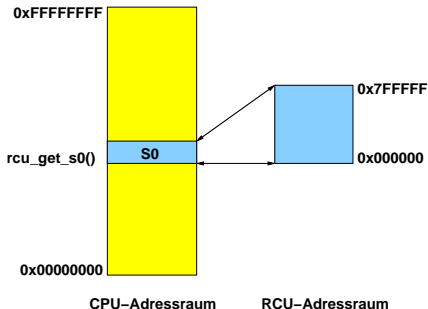


- “Herz” des Systems rekonfigurierbar
- RCU als Koprozessor mit CPU auf einem Chip

Memory-Mapped RCU



A. Koch



- Basisadresse der RCU mit `rcu_get_s0()` abrufen
- Zugriffe in diesen Speicherbereich landen auf RCU
 - Fangen dort bei RCU-Adresse 0 an
 - Annahme hier: RCU Bereich hat 8MB, als 2M x 32b Worte adressierbar

Kommunikation im Slave-Mode



A. Koch

- CPU füttert RCU mit Daten und Kontrollinformationen
- RCU rechnet
- CPU holt Ergebnisse von RCU ab
- RCU agiert als **Sklave** im System
- Einfach zu realisieren, aber langsam

Slave-Mode Schnittstelle im ML310 ACS



A. Koch

- input **CLK**
 - Taktsignal
- input **RESET**
 - Bei 1: Alle internen Register auf Startwerte setzen
- input **ADDRESSED**
 - Bei 1: Ein Zugriff von aussen auf die RCU erfolgt
- input **WRITE**
 - Bei 1: Schreibzugriff von aussen, Daten liegen an RCU an
 - Bei 0: Lesezugriff, Daten aus der RCU nach aussen geben
- input **DATAIN**
 - Bei Schreibzugriff von hier Daten übernehmen
- input **ADDRESS**
 - Zugegriffene Adresse innerhalb des RCU-Bereiches
- output **DATAOUT**
 - Bei Lesezugriff hier Daten anlegen

The diagram illustrates a 32-bit input-output permutation. The input, labeled 'Eingabewort', consists of bits 31, 30, 29, 28, followed by six dots, then bits 3, 2, 1, and 0. The output, labeled 'Ausgabewort', also consists of bits 31, 30, 29, 28, followed by six dots, then bits 3, 2, 1, and 0. Arrows show a bit-reversed mapping: input bit 31 connects to output bit 0, input bit 30 to output bit 1, input bit 29 to output bit 2, input bit 28 to output bit 3, and so on, with input bit 3 connecting to output bit 28 and input bit 0 connecting to output bit 31.

- 16 / 46

Slave-RCU in Verilog

Schnittstelle zum Restsystem



```
module user(  
    CLK,           // Systemtakt  
    RESET,        // systemweiter Reset  
    ADDRESSED,    // High, wenn RCU von CPU angesprochen wird  
    WRITE,        // High, wenn CPU auf RCU schreiben will  
    DATAIN,      // Von der CPU auf die RCU geschriebene Daten  
    DATAOUT,     // Von der CPU aus der RCU gelesene Daten  
    ADDRESS // Adresse des Zugriffs (hier ignoriert)  
);  
  
    // Eingänge  
input      CLK;  
input      RESET;  
input      ADDRESSED;  
input      WRITE;  
input [31:0] DATAIN;  
input [23:2] ADDRESS;  
  
    // Ausgänge  
output [31:0] DATAOUT;
```

A. Koch

Slave-RCU in Verilog

Kern der Anwendung



A. Koch

```
reg [31:0] result;           // Ergebnisregister
reg [31:0] reversed;         // Zwischenergebnis

// Gebe immer (unabhängig von der Adresse) das Ergebnisregister aus
assign DATAOUT = result;

// Berechne als Zwischenergebnis immer die
// bitverdrehte Reihenfolges des Dateneingangs
// Beachte: Dies ist ein kombinatorischer Block!
always @(DATAIN) begin: comb_block
    integer n;
    for (n=0; n < 32; n = n + 1) begin
        reversed[n] = DATAIN[31-n];
    end
end

// Steuerung
always @(posedge CLK or posedge RESET) begin
    // Initialisiere Ergebnis auf magic number für Debugging
    if (RESET) begin
        result <= 32'hDEADBEEF;
    // Schreibzugriff auf RC, neu berechnetes Zwischenergebnis übernehmen
    end else if (ADDRESSED & WRITE) begin
        result <= reversed;
    end
end

endmodule
```

Programm zur Benutzung der Slave-RCU

Kernteil ohne Initialisierung und Deinitialisierung



A. Koch

```
// Lese komplette Eingabedatei in Eingabe-Speicherbereich  
fread(inwords, sizeof(unsigned long), NUM_WORDS, infile);
```

```
// RCU initialisieren  
rcu_init ();
```

```
// Zeiger auf RC-Adressraum holen  
rcu = rcu_get_s0(NULL);
```

```
// Bearbeite Daten  
for (m=0; m < NUM_WORDS; ++m) {  
    // Übertrage das Eingabedatenwort an die RCU  
    rcu[0] = inwords[m];  
    // Hole das Ergebnis von der RCU in Ausgabefeld ein  
    outwords[m] = rcu[0];  
}
```

```
// Schreibe das komplette Ausgabefeld in die Ausgabedatei  
fwrite (outwords, sizeof(unsigned long), NUM_WORDS, outfile);
```



A. Koch

Demo



A. Koch

Master-Mode



- Funktioniert zwar
- Aber recht langsam
- Bus wird nicht effizient ausgenutzt
- Immer nur ein 32b Wort von/nach CPU übertragen
- Dazwischen längere Pausen

➡ Besser: **Master-Mode**

Master-Mode



A. Koch

- Erwartet nur **Kommandos** und **Parameter** im Slave-Mode
- Nicht mehr die **Daten** selbst
- Arbeitet nach Startkommando unabhängig von CPU
- Signalisiert CPU später das Ende der Berechnung
- Problem: Master-Mode-Protokolle nicht trivial!

➡ Abstrahieren und wiederverwendbare Lösung finden



A. Koch

Abstrakte Datenströme mit MARC

Idee von MARC

Memory Architecture for Reconfigurable Computers



A. Koch

- Abstrahiert echte ACS-Hardware
- Eigentliche Anwendung bleibt portabel
 - ACE-V ACS
 - ML310 ACS (aber $\gg 18x$ schneller!)
- Charakterisiert **logische** Zugriffsmuster
 - Irregulär: Z.B. Durchlauf von verketteter Liste
 - Regulär: Z.B. Durchlauf durch Array
- Hier: Reguläre Zugriffe mit **Datenströmen** (*streams*)

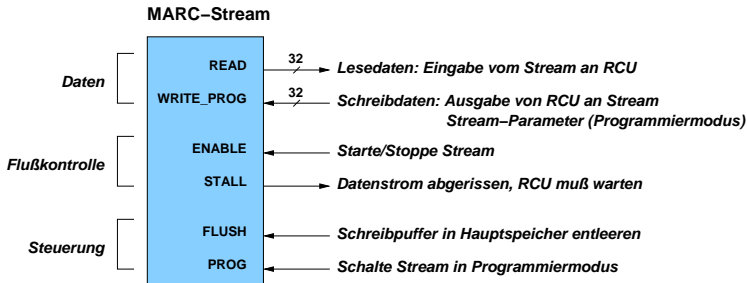


- Laufen über einen Speicherbereich
 - Liefern Daten an RCU (Lese-Strom)
 - Akzeptieren Daten von RCU (Schreib-Strom)
- Parameter
 - Startadresse
 - Anzahl Datensätze -1
 - Schrittweite (übersprungene Datensätze)
 - Breite der Zugriffe (8b/16b/32b)
 - Lesen oder Schreiben
- Signalisieren Unterbrechungen (*stalls*) im Datenstrom
 - RCU muss reagieren
 - Beim Lesen: Warten, bis wieder Daten verfügbar sind
 - Beim Schreiben: Anhalten und Datum puffern, bis Strom weiterfließt
- Problem: Gekoppelte Ströme

Schnittstelle zu MARC-Stream



A. Koch



Reihenfolge der Parameter im Programmiermodus

Startadresse

Anzahl Datensätze

Schrittweite

Breite der Zugriffe (8b/16b/32b)

Zugriffsart (Lesen/Schreiben)

Flußsteuerung mit **ENABLE**-Eingang

Kommando wird **vor** positiver Taktflanke angelegt



Beim Lesen

- **1**: RCU will zur **übernächsten** Flanke ein neues Datum übernehmen
- **0**: Stream anhalten, an **übernächster** Flanke kein Datum übernehmen

Beim Schreiben

- **1**: RCU hat gültiges Datum berechnet, das zur **nächsten** Flanke in Strom übernommen werden soll.
- **0**: RCU hat für die **nächste** Flanke noch kein gültiges Datum, Strom soll warten

A. Koch

Flußsteuerung mit **STALL**-Ausgang

Zeigt durch '1' eine Unterbrechung im Datenfluß an



A. Koch

Beim Lesen

- Daten vom Strom nur in RCU übernehmen, wenn zur **vorherigen** positiven Taktflanke **STALL=0** ist
- Bei **STALL=1** auf Daten warten

Beim Schreiben

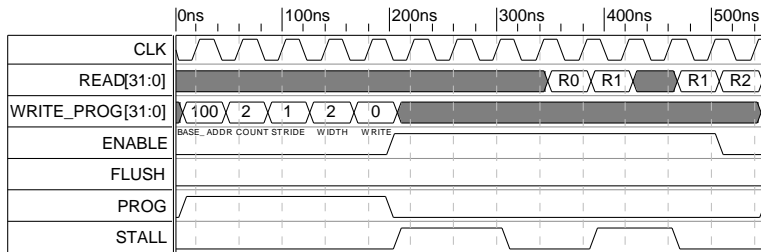
- Strom übernimmt Daten nur, wenn zur **selben** positiven Taktflanke **STALL=0** ist
- Bei **STALL=1** Schreibdatum stabil halten und auf Ende der Unterbrechung warten

Lesestrom

Programmierung und Behandlung von Unterbrechung



A. Koch



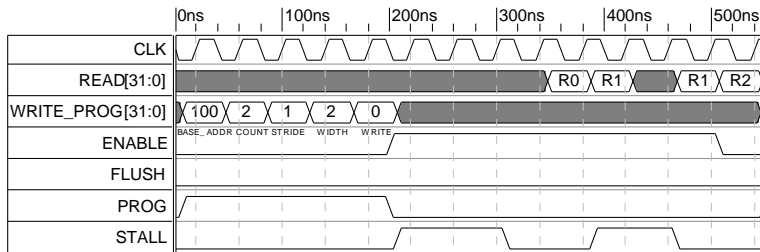
1 Takt **Versatz** zwischen Steuersignalen und Daten

Schreibstrom

Programmierung und Behandlung von Unterbrechung



A. Koch



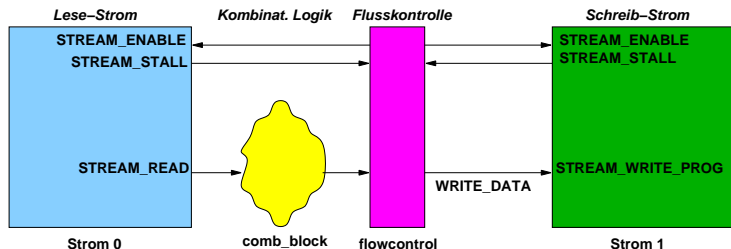
Kein Versatz zwischen Steuersignalen und Daten



- Abrisse
 - **Lesestrom** hat temporär keine Daten verfügbar
 - **Schreibstrom** kann temporär keine Daten aufnehmen
- Jeweils anderer Strom muss **angehalten** werden
- Beim **Schreibstrom** zusätzlich
 - Letztes Datum **zwischenspeichern**
 - Leseströme reagieren mit 1 Takt Verzögerung auf Kommandos
 - Dieses Datum auffangen
 - Gespeichertes Datum beim Wiederaufstart des Schreibstroms übergeben

Struktur einer RCU mit zwei Strömen

Lesestrom gekoppelt mit Schreibstrom



A. Koch

- Berechnung als **kombinatorische** Logik
 - Arbeitet innerhalb **1** Takt
- Flußkontrolle synchronisiert Ströme und puffert Daten
- Bei komplizierterer **sequentieller** Berechnung
 - Steuersignale verzögern
 - Berechnung anhalten
 - Falls nicht möglich: Mehr Daten puffern



A. Koch

Programmbeispiel Master-Mode-RCU

Schnittstelle der Master-Mode-RCU

Erweitert Slave-Schnittstelle um MARC-Signale



```
'include "marcdefs.v"
```

```
module user (
```

```
    STREAM_READ,    // Read-Datenbus  
    STREAM_WRITE_PROG, // Write-Programm-Datenbus  
    STREAM_STALL,    // Stall-Signale  
    STREAM_ENABLE,   // Start/Stop für Streams  
    STREAM_FLUSH,    // Schreib-Streams entleeren  
    STREAM_PROG      // Programmiermodus einschalten
```

```
// Deklarationen für Stream-Schnittstelle
```

```
input ['STREAM_DATA_BUS] STREAM_READ;  
input ['STREAM_CNTL_BUS] STREAM_STALL;  
output ['STREAM_DATA_BUS] STREAM_WRITE_PROG;  
output ['STREAM_CNTL_BUS] STREAM_ENABLE;  
output ['STREAM_CNTL_BUS] STREAM_FLUSH;  
output ['STREAM_CNTL_BUS] STREAM_PROG;  
wire ['STREAM_DATA_BUS] STREAM_READ;  
wire ['STREAM_DATA_BUS] STREAM_WRITE_PROG;  
wire ['STREAM_CNTL_BUS] STREAM_STALL;  
wire ['STREAM_CNTL_BUS] STREAM_ENABLE;  
reg ['STREAM_CNTL_BUS] STREAM_FLUSH;  
reg ['STREAM_CNTL_BUS] STREAM_PROG;
```

A. Koch

Slave-Mode-Schnittstelle

Zum Übertragen von Kommandos und Parametern



A. Koch

```
// Abkürzung für Registernummer 0 ... 15
wire [3:0] REGNUM = ADDRESS[5:2];

// Gebe immer das gerade adressierte Register aus.
// Nicht benötigte Register geben eine Magic-Number
// und den aktuellen IRQ-Status im MSB zurück
wire [31:0] DATAOUT =
    (REGNUM == 4'h0) ? SOURCEADDR
    : (REGNUM == 4'h1) ? DESTADDR
    : (REGNUM == 4'h2) ? COUNT
    : (32'h00C0FFEE | (IRQSTATE << 31));

// Controller FSM überwacht gesamte Anwendung
always @(posedge CLK or posedge RESET) begin
    // Initialisiere Register bei chip-weitem Reset
    if (RESET) begin
end else if (ADDRESSED & WRITE) begin
    // Schreibzugriff auf RCU, schreibe in entsprechendes Register
    case (REGNUM)
        0: SOURCEADDR <= DATAIN;
        1: DESTADDR <= DATAIN;
        2: COUNT <= DATAIN;
        3: begin
            START <= 1; // Startkommando, beginne Ausführung
        end
        default: ;
    endcase
```

1. Betriebsphase: Programmieren der Ströme



A. Koch

```
end else begin
  // CPU hat Berechnung gestartet, keine Slave-Mode Zugriffe mehr möglich
  if (START) begin
    case (STATE)
      'STATE_PROG.START:
        begin
          // Beide Streams in Programmiersmodus schalten
          STREAM_PROG[1:0] <= 2'b11;
          // Anfangsadresse für Stream 0 schreiben
          STREAM_PROGDATA_0 <= SOURCEADDR;
          // Anfangsadresse für Stream 1 schreiben
          STREAM_PROGDATA_1 <= DESTADDR;
          // FSM weitersetzen
          STATE <= 'STATE_PROG.COUNT;
        end
      'STATE_PROG.COUNT:
        begin
          // Anzahl Datensätze - 1 eintragen (bei beiden Streams gleich)
          STREAM_PROGDATA_0 <= COUNT - 1;
          STREAM_PROGDATA_1 <= COUNT - 1;
          // FSM weitersetzen
          STATE <= 'STATE_PROG.STEP;
        end
    end
  end
end
```

Analog weiter für Schrittweite, Zugriffsbreite, Betriebsart
(Lesen/Schreiben)

2. Betriebsphase: Fließende Datenströme

Während der eigentlichen Berechnung



A. Koch

```
'STATE_COMPUTE:
  begin
    // Programmiermodus für beide Streams abschalten
    STREAM_PROG[1:0] <= 0;
    // Beide Streams starten (via flowcontrol-Modul)
    STREAMSTART <= 1;

    // Alle Datensätze bearbeitet?
    if (COUNT == 0) begin
      // Dann beide Streams stoppen
      STREAMSTART <= 0;
      // Falls Schreib-Stream fertig
      if (!STREAM_STALL[1]) begin
        // alle noch gepufferten Daten wirklich schreiben
        STREAM_FLUSH[1] <= 1;
        // FSM weiterrsetzen
        STATE <= 'STATE_SHUTDOWN;
      end
    end else if (STREAM_ENABLE[0] & ~STREAM_STALL[0])
      // Nur dann einen Datensatz als bearbeitet zählen,
      // wenn Stream 0 aktiv liest (ENABLE) und nicht hängt (!STALL)
      COUNT <= COUNT - 1;
  end
```

3. Betriebsphase: Herunterfahren der Ströme



A. Koch

```
'STATE_SHUTDOWN:
  begin
    // Ist Schreibpuffer schon komplett geleert?
    if (!STREAM_STALL[1]) begin
      // ja, Leerung beenden
      STREAM_FLUSH[1] <= 0;
      // CPU durch IRQ Fertigwerden der RC anzeigen
      IRQSTATE <= 1;
      // FSM stoppen (RC jetzt wieder im Slave-Mode)
      START <= 0;
      // FSM auf Startzustand zurücksetzen
      STATE <= 'STATE_PROG_START;
    end
  end
  // Dieser Fall sollte nicht auftreten, nur für Logikoptimierung
  default: STATE <= 'bx;
endcase
```

Verbinden der Datenströme



A. Koch

```
// Streams laufen, nachdem sie gestartet worden sind und solange
// noch Daten zu bearbeiten sind.
wire RUNNING = STREAMSTART & (COUNT != 0);

// Flußkontrolle zwischen Ein- und Ausgabe-Streams
flowcontrol FC (
    CLK,           // Takt
    RUNNING,       // Streams laufen lassen?
    STREAM_STALL[0], // Hängt Stream 0 (Eingabe-Stream)?
    STREAM_STALL[1], // Hängt Stream 1 (Ausgabe-Stream)?
    REVERSED,      // Von Anwendung zu schreibende Daten
    STREAM_ENABLE[0], // Stream 0 starten oder anhalten
    STREAM_ENABLE[1], // Stream 1 starten oder anhalten
    WRITE_DATA     // Eingangsdaten für Ausgabe-Stream
);

// Schalte Streams zwischen Programmier- und Datenbetrieb um
// Stream0 ist Lese-Stream, sein Eingang kann immer im Programmierbetrieb sein
assign STREAM_WRITE_PROG[STREAM_0] = STREAM_PROGDATA_0;

// Stream1 ist Schreib-Stream, hier muß der Eingang umgeschaltet werden
assign STREAM_WRITE_PROG[STREAM_1] =
    (STREAM_PROG[1])
    ? STREAM_PROGDATA_1
    : WRITE_DATA;
```


Eigentliche Berechnung



A. Koch

```
// Berechne als Zwischenergebnis immer die
// bitverdrehte Reihenfolge des Lesestromes 0
// Beachte: Dies ist ein kombinatorischer Block!
always @(STREAM_READ[31:0]) begin: comb_block
    integer n;
    for (n=0; n < 32; n = n + 1) begin
        REVERSED[n] = STREAM_READ[31-n];
    end
end
```

Programm zur Benutzung der Master-RCU: Initialisierung



A. Koch

```
// RC initialisieren
rcu_init ();

// Zeiger auf S0-Bereich mit RC-Registern holen
rcu = rcu_get_s0(NULL);

// fordere Speicher für Ein- und Ausgabefelder an
// Platz für Ausgabe liegt direkt hinter Eingabefeld
// *.phys zeigt auf die physikalische Speicheradresse
// aus Sicht der Hardware
inwords = acev_malloc_master(2 * NUM_WORDS * sizeof(unsigned long),
                             (void **) &inwords_phys);
outwords = inwords + NUM_WORDS;
outwords_phys = inwords_phys + NUM_WORDS;

if (!inwords || !inwords_phys) {
    fprintf (stderr, "out_of_memory\n");
    exit (1);
}

// Funktioniert der Slave Zugriff?
printf ("Magic:_%x\n", rc[28]);
```

Programm zur Benutzung der Master-RCU: Start und Ende



A. Koch

```
// Lese komplette Eingabedatei in Eingabe-Speicherbereich
fread(inwords, sizeof(unsigned long), NUM.WORDS, infile);

// Übertrage Parameter an RCU (nicht die Daten selbst)
rc[REG_SOURCE_ADDR] = inwords_phys; // Physikalische(!) Startadresse
rc[REG_DEST_ADDR] = outwords_phys; // Physikalische(!) Zieladresse
rc[REG_COUNT] = NUM.WORDS; // Anzahl Datensätze
rc[REG_START] = 1; // Startkommando für RC

// Warte auf Ende der Berechnung (wird über IRQ angezeigt)
rcu_wait ();

// Schreibe das komplette Ausgabefeld in die Ausgabedatei
fwrite (outwords, sizeof(unsigned long), NUM.WORDS, outfile);
```



Lösung	Taktfreq. in MHz	RCU-Größe in Slices	Rechenzeit in μ s	Schneller als SW um
Software	300 (CPU)		195942	1.00
Slave-RCU	100 (RCU)	45	32045	6.11
Master-RCU	100 (RCU)	981	5813	33.71

Ziel-RCU: Virtex-IIpro FPGA, 4 LUTs pro Slice



A. Koch

Zusammenfassung



- Grundlagen Verilog
 - Kommunikation der RCU mit Restsystem
 - Slave-Mode
 - Master-Mode
 - Beispielanwendung: Bit-Reihenfolge verdrehen
 - Datenströme mit MARC
 - Flußsteuerung und Synchronisation
- ➡ Detailliertere Erklärung in **Teilskript** auf Web-Seite