4. Entwurf des VISCY-Prozessors
(Very reduced Instruction Set Computer sYstem)

4.1. Überblick

- **Architektur der VISCY-CPU**
  - RISC-Architektur
  - einfache Operationen
  - einfache Befehlscodierung: ein Wort pro Befehl (einheitlich)
  - Load-/Store-Architektur
  - 3-Adress-Befehle
  - 8 allgemeine Register: r0, ..., r7
  - Wortbreite 16-Bit
  - kein Status-Register (bedingte Sprünge durch Test, ob Register = 0)
  - kein Prozessor-Stack (kann per Software realisiert werden)

-> Befehlssatz ist einfach, aber dennoch (Turing-)vollständig!

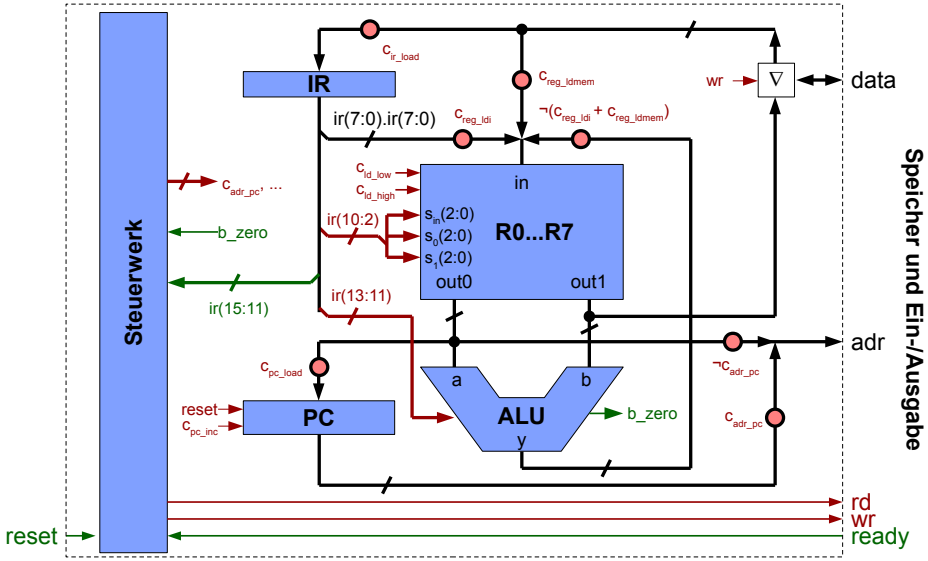
Befehlssatz und -codierung

Operation	Opcode	Bedeutung
ADD	00 000 ddd sss ttt --	Rddd <- Rsss + Rttt
SUB	00 001 ddd sss ttt --	Rddd <- Rsss - Rttt
SAL	00 010 ddd sss --- --	Rddd <- Rsss(14:0).'0'
SAR	00 011 ddd sss --- --	Rddd <- Rsss(15).Rsss(15:1)
AND	00 100 ddd sss ttt --	Rddd <- Rsss & Rttt
OR	00 101 ddd sss ttt --	Rddd <- Rsss | Rttt
XOR	00 110 ddd sss ttt --	Rddd <- Rsss ^ Rttt
NOT	00 111 ddd sss --- --	Rddd <- ~Rsss
LDIL	01 -00 ddd nnn nnn nn	Rddd[7:0] <- nnnnnnnn
LDIH	01 -01 ddd nnn nnn nn	Rddd[15:8] <- nnnnnnnn
LD	01 -10 ddd sss --- --	Rddd <- Mem[Rsss]
ST	01 -11 --- sss ttt --	Mem[Rsss] <- Rttt
JMP	10 -00 --- sss --- --	PC <- Rsss
HALT	10 -01 --- --- --- --	Prozessor hält an
JZ	10 -10 --- sss ttt --	if (Rttt == 0) PC <- Rss
JNZ	10 -11 --- sss ttt --	if (Rttt != 0) PC <- Rss

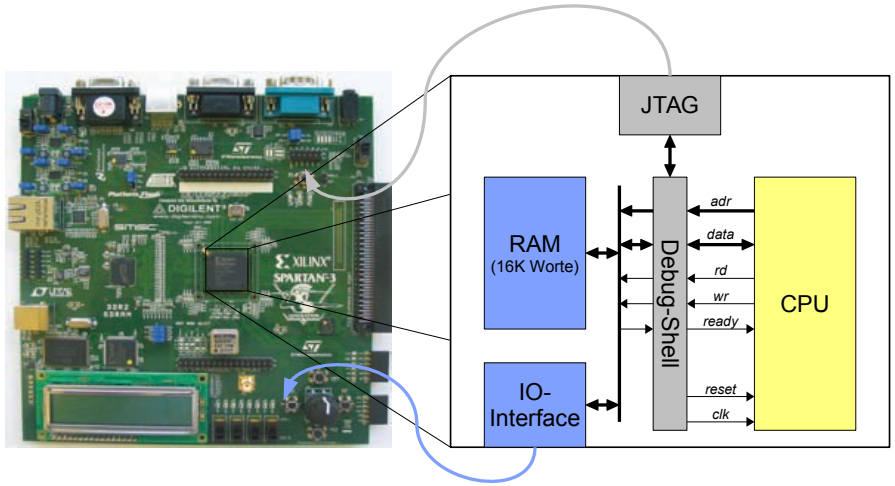
Beispiel-Programm

```
; Berechnung der ersten 8 Fibonacci-Zahlen

        .org 0x0000           ; alles folgende ab Adresse 0
start:  xor  r0, r0, r0       ; r0 := 0
        ldil r1, 1            ; r1 := 1
        ldih r1, 0
        ldil r2, result & 255 ; r2 := result (Adresse)
        ldih r2, result >> 8
        ldil r3, 8            ; r3 := 8 (Schleifenzähler)
        ldih r3, 0
        ldil r4, loop & 255   ; r4 := loop (Sprungadresse)
        ldih r4, loop >> 8
        xor  r5, r5, r5       ; r5 := 0 = fib(0)
        add  r6, r0, r1       ; r6 := 1 = fib(1)
loop:   st   [r2], r5         ; aktuelle Fibonacci-Zahl schreiben
        add  r7, r5, r6       ; r7 := r5 + r6 = fib(n) + fib(n+1)
        or   r5, r6, r0       ; r5 := r6
        or   r6, r7, r0       ; r6 := r7
        add  r2, r2, r1       ; Zieladresse erhöhen
        sub  r3, r3, r1       ; r3 := r3 - 1, Schleifenzähler erniedrigen
        jnz  r3, r4           ; Sprung nach 'loop', falls r3 != 0
        halt                  ; Prozessor anhalten

        .org 0x0100           ; alles folgende ab Adresse 0x0100
result: .res 8                ; 8 Worte reservieren
        .end
```

Struktur der CPU



Umgebung des VISCY-Prozessors



Schnittstelle der CPU

```
entity CPU is
  port (
    clk, reset: in std_logic;
      -- Takt & Reset
    adr: out std_logic_vector (15 downto 0);
      -- Adressbus
    data: inout std_logic_vector (15 downto 0);
      -- Datenbus
    rd, wr: out std_logic;
      -- Lese- und Schreibanforderung
    ready: in std_logic
      -- Rückmeldung für Lese-/Schreibzugriffe
  );
end CPU;
```


Praktikum: Zwei Varianten

1. Entwicklung des Prozessors ("blaue Piste")

- **Sprache:** VHDL
- **Ziel:**
  - Entwicklung der VISCY-CPU und Inbetriebnahme auf einem FPGA-Board

2. Entwicklung eines System-Modells für Leistungsanalysen ("rote Piste")

- **Sprache:** SystemC
- **Ziel:**
  - Entwicklung eines taktgenauen System-Modells in SystemC
  - Möglichkeit zur Leistungsanalyse auf System- und Architektur-Ebene (z.B. Auswirkung von Architekturänderungen, Speichereffizienz, Mehrkern-Varianten)

Ablauf des Praktikums (VHDL-Variante): Drei Phasen

1. Entwurf der Teilkomponenten

- Kennenlernen der Entwurfs-Software
- Simulation und Synthese der Komponenten

2. Entwurf des Steuerwerks und der Gesamtstruktur

- Entwurf des Steuerwerkes (zunächst nur teilweise)
- Simulation der gesamten CPU
- Entwicklung der Software

3. Inbetriebnahme und Optimierung

4.2. Entwurf der Teilkomponenten

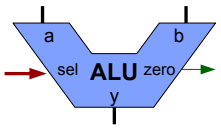
a) ALU
b) Register-File
c) Programmzähler (PC)
d) Befehlsregister (IR)

Arbeitsschritte:

- Entwurf der Komponente (synthetisierbares VHDL, Architektur "RTL")
- Simulation mit Testbench
- Synthese

a) ALU

```
entity ALU is
  port (
    a   : in std_logic_vector (15 downto 0);
    b   : in std_logic_vector (15 downto 0);
    sel : in std_logic_vector (2 downto 0);
    y   : out std_logic_vector (15 downto 0);
    zero: out std_logic
  );
end ALU;
```



• Funktion

- Ausführen aller ALU-Operationen
- Erzeugen des *zero*-Signals für bedingte Sprünge
- Operationen und deren Codierung: siehe Befehlssatz!

• Checkliste zur Testbench

- Funktionieren die Operationen auf **kompletter Wortbreite?**
  Tipp: einzelne ausgewählte + größere Menge (pseudo-)zufällige Muster simulieren
- Sind die Operationen richtig **codiert**? (oder wird z.B. oder-verknüpft statt addiert?)
  Tipp: ALU-Entwurf und Testbench unabhängig voneinander eingeben (verschiedene Personen!)
- Funktioniert der ***zero*-Ausgang**?
- Funktionieren die **Schiebe-Operationen** richtig (**arithmetisch**!)

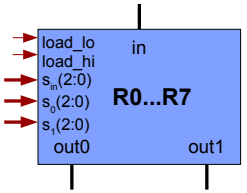
Tipp: Schreiben Sie eine Prozedur, die zwei Operanden *a, b* entgegennimmt und mit ihnen alle 8 Operationen durchspielt und die Ausgaben automatisch überprüft.

• Hinweise zur Synthese

- Die ALU muss vollständig kombinatorisch sein.
- Synthese-Report beachten:
  - Wurden Latches oder Flipflops erzeugt?
  - Welche Komponenten wurden erzeugt? Entspricht das den Erwartungen, oder wurde evtl. unnötige Logik generiert?
  - Welche Warnungen / Fehlermeldungen gibt es?

b) Register-File

```
entity REGFILE is
  port (
    clk: in std_logic;
    in_data: in std_logic_vector (15 downto 0);
    in_sel: in std_logic_vector (2 downto 0);
    out0_data: out std_logic_vector (15 downto 0);
    out0_sel: in std_logic_vector (2 downto 0);
    out1_data: out std_logic_vector (15 downto 0);
    out1_sel: in std_logic_vector (2 downto 0);
    load_lo, load_hi: in  std_logic
  );
end REGFILE;
```



• Funktion

- Enthält die allgemeinen Register R0, ..., R7
- Zwei unabhängige Ausgabe-Ports (-> zwei ALU-Operanden)
- Ein Eingabe-Bus
- Low- und High-Byte können unabhängig voneinander geladen werden (-> LDIL-/LDIH-Befehle)

• Tipps zum Entwurf

- Für Register-Inhalte Array verwenden
- Konvertieren der Select-Leitungen in Register-Nr. mit *conv_integer* möglich

```
...
architecture RTL of REGFILE is
  type t_regfile is array (0 to 7) of STD_LOGIC_VECTOR(15 downto 0);
  signal reg: t_regfile;
begin

  -- Ausgabe...
  out0_data <= reg(to_integer (unsigned (out0_sel)));
  ...

  -- Zustandsübergang...
  process (clk)
  begin
    if rising_edge (clk) then
      if (load_lo = '1') then
        reg(to_integer (unsigned (in_sel))) (7 downto 0) <= in_data(7 downto 0);

    ...
  end process;

end RTL;
```


• Aus einer Testbench ...

```
-- Reset: Alle Register auf 0 setzen...
run_cycle;
in_data <= "0000000000000000";
load_hi <= '1'; load_lo <= '1';
for i in 0 to 7 loop
  in_sel <= std_logic_vector (
              to_unsigned (i, 3));
  run_cycle;
end loop;

-- nur High-Bytes schreiben...
in_data <= "1111111100000000";
load_hi <= '1'; load_lo <= '0';
for i in 0 to 7 loop
  in_sel <= std_logic_vector (
              to_unsigned (i, 3));
  run_cycle;
end loop;
load_hi <= '0';

-- Registerinhalte lesen & überprüfen...
for i in 0 to 7 loop
  out0_sel <= std_logic_vector (to_unsigned (i, 3);
  out1_sel <= std_logic_vector (to_unsigned (i, 3);
  run_cycle;
  assert out0_data = "1111111100000000";
  assert out1_data = "1111111100000000";
end loop;
...
```

```
...

-- jetzt Low-Bytes schreiben...
in_data <= "1111111111111111";
load_hi <= '0'; load_lo <= '1';
for i in 0 to 7 loop
  in_sel <= std_logic_vector (
              to_unsigned (i, 3));
  run_cycle;
end loop;
load_lo <= '0';

-- Registerinhalte lesen & überprüfen...
for i in 0 to 7 loop
  out0_sel <= std_logic_vector (
                to_unsigned (i, 3));
  out1_sel <= std_logic_vector (
                to_unsigned (i, 3));
  run_cycle;
  assert out0_data = "1111111111111111";
  assert out1_data = "1111111111111111";
end loop;

...
```

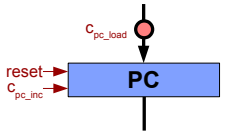
- **Checkliste für die Testbench**
  - Wo wird erkannt, ob
    - irgendwo in das falsche Register geschrieben wird?
    - irgendwo aus dem falschen Register gelesen wird?
    - bei *load_hi* = 0 bzw. *load_lo* = 0 das entsprechende Halbwort gehalten wird, egal
      - was in dem Halbwort gespeichert ist?
      - was am Dateneingang anliegt?
      - was in das andere Halbwort geschrieben wird?

- **Tipps:**
  - verschiedene Werte in verschiedene Register schreiben
  - getrennte Schleifen zum Schreiben/Lesen verwenden
  - *out0_sel* und *out1_sel* unterschiedlich belegen
  - Testbench mit absichtlich falschem Entwurf überprüfen!

- **Hinweise zur Synthese**
  - Das Register-File soll exakt 128 (=8 * 16) D-Flipflops enthalten
    - Keine pegelgetriggerten Latches!
    - Keine weiteren D-Flipflops
  - Synthese-Report beachten:
    - Wo wurden Latches oder Flipflops erzeugt?
    - Welche Komponenten wurden erzeugt? Entspricht das den Erwartungen, oder wurde evtl. unnötige Logik generiert?
    - Welche Warnungen / Fehlermeldungen gibt es?

c) Programmzähler

```
entity PC is
  port (
    clk: in std_logic;
    reset, inc, load: in std_logic;
    pc_in: in std_logic_vector (15 downto 0);
    pc_out: out std_logic_vector (15 downto 0)
  );
end PC;
```



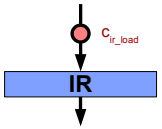
- **Funktionalitäten**
  - Zurücksetzen (Programmstart an Adresse 0 nach globalem Reset)
    - Reset dominiert gegenüber allen anderen Operationen
  - Wert halten
  - Inkrementieren (nächster Befehl)
  - Wert laden (Sprungbefehle)

- **Checkliste zur Testbench**
  - Funktioniert der Reset und dominiert er?
  - Funktioniert das Laden?
  - Funktioniert das Inkrementieren über die gesamte Worbreite?
  - Funktioniert das Halten von Werten?

- **Synthese**
  - siehe oben
  - Sequentielle Elemente: 16 D-Flipflops (sonst nichts!)

d) Befehlsregister

```
entity IR is
  port (
    clk: in std_logic;
    load: in std_logic;
    ir_in: in std_logic_vector (15 downto 0);
    ir_out: out std_logic_vector (15 downto 0)
  );
end IR;
```



- **Funktionalitäten**
  - Wert halten
  - Wert laden (neuer Befehl)
- **Checkliste zur Testbench**
  - Funktioniert das Laden und Halten?
- **Synthese**
  - Sequentielle Elemente: 16 D-Flipflops (sonst nichts!)

4.3. Steuerwerk & Zusammenbau der CPU

4.3.1. CPU-Gesamtstruktur

- **Komponente 'CPU' enthält:**
  - bisher entworfene Komponenten
  - Verbindungsstrukturen
  - Steuerwerk

=> Struktur-Stil

mit kleineren Ausnahmen (Verbindungen)

Aufbau der Architektur (Muster)

```
architecture RTL of CPU is

  -- Component declarations...
  component alu is ...

  -- Configuration...
  for all: ALU use entity WORK.ALU(RTL);
  ...
  -- Internal signals...
  signal  alu_y, regfile_out0_data, regfile_out1_data, regfile_in_data,
          mem_data_in, mem_data_out: STD_LOGIC_VECTOR (15 downto 0);
  signal  alu_zero: STD_LOGIC;
  ...
begin
  -- Component instatiations...
  U_ALU: ALU port map (
      a => regfile_out0_data, b => regfile_out1_data, y => alu_y, sel => ir(13 downto 11),
      zero => alu_zero
    );
  ...

  -- Multiplexer vor Adressbus...
  process (pc, regfile_out0_data, c_adr_pc_not_reg)
  begin
    if c_adr_pc_not_reg = '1' then  adr <= pc;
    else                            adr <= regfile_out0_data;
    end if;
  end process;

...
```


4.3.2. Tristate-Busse in VHDL

- Hat ein Signal mehr als einen ***Treiber***, so wird der resultierende Wert mit einer ***Auflösungsfunktion*** *("resolution function")* ermittelt (implizit!).

  `0 → 0 ← Z`
  `Z → 1 ← 1`
  `Z → Z ← Z`
  `0 → X ← 1` (Buskonflikt)

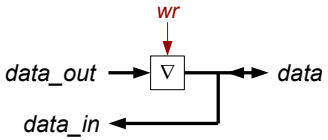
- ***Treiber*** können sein:
  - Ports vom Typ '*in*' oder '*inout*'
  - parallele Signalzuweisungen
  - instanziierte Komponenten
  - Prozesse

- Bei mehreren Zuweisungen in einem Prozess wird keine Auflösungsfunktion verwendet (letzte Zuweisung setzt sich durch).

Datenbus *'data'*

Umsetzung des Tristate-Busses auf zwei gerichtete Busse

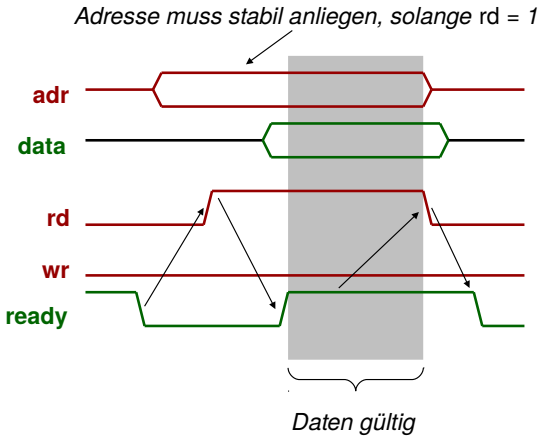
```
entity CPU is
  port ( ...
      data: inout std_logic_vector (15 downto 0);
      ...
    );
end CPU;

architecture RTL of CPU is
  signal mem_data_in, mem_data_out: std_logic_vector (15 downto 0);
  ...
begin
  ...
  process (mem_data_out, c_mem_wr) -- Prozess für Ausgabe
  begin
    if c_mem_wr = '1' then
      data <= mem_data_out;        -- CPU treibt den Bus
    else
      data <= "ZZZZZZZZZZZZZZZZ";  -- CPU verhält sich passiv
    end if;
  end process;

  mem_data_in <= data;             -- Dateneingang
  ...
end RTL;
```



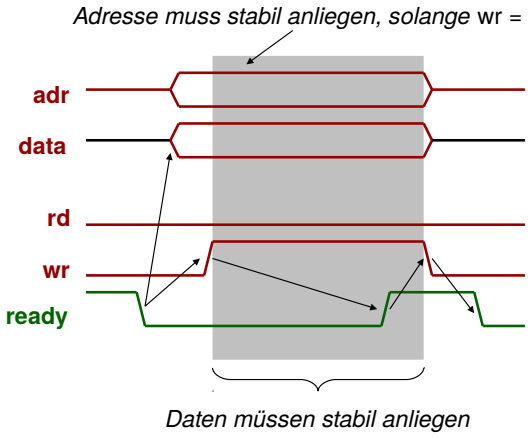
4.3.3. Speicher-Schnittstelle

- **Signale**
  - `adr: out std_logic_vector (15 downto 0)`
    - Adressbus (CPU -> Speicher)
  - `data: inout std_logic_vector (15 downto 0)`
    - Datenbus (bidirektional)
  - `rd, wr: out std_logic`
    - Schreib-/Leseanforderung der CPU (-> '1')
    - Zeigt an, dass Daten übernommen (-> '0')
  - `ready: in std_logic`
    - Zeigt an, dass Speicher bereit für neue Anforderung (-> '0')
    - Zeigt an, dass Speicher Transaktion ausgeführt hat (-> '1')

Speicher-Lesezyklus



Speicher-Schreibzyklus



4.3.4. Steuerwerk

Entwurfsschritte:

a) Ein-/Ausgänge bestimmen

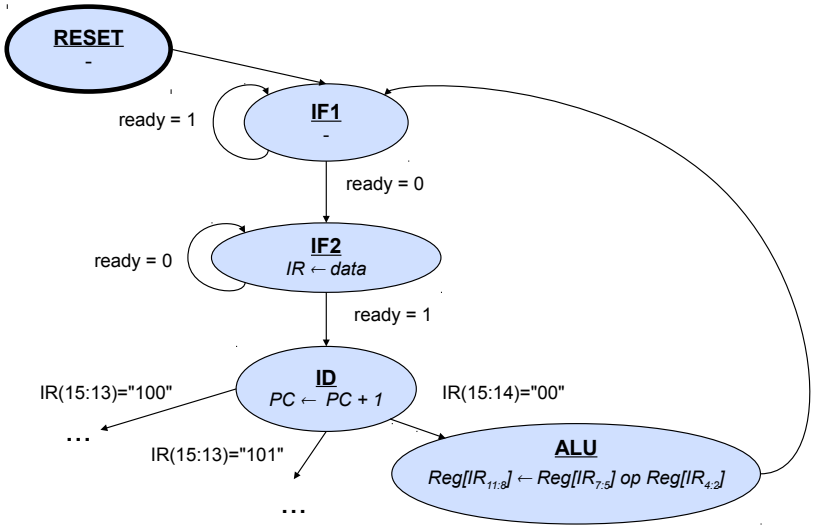
b) Zustandsübergangs-Diagramm entwerfen

c) Steuerwerk in VHDL modellieren

a) Ein-/Ausgänge

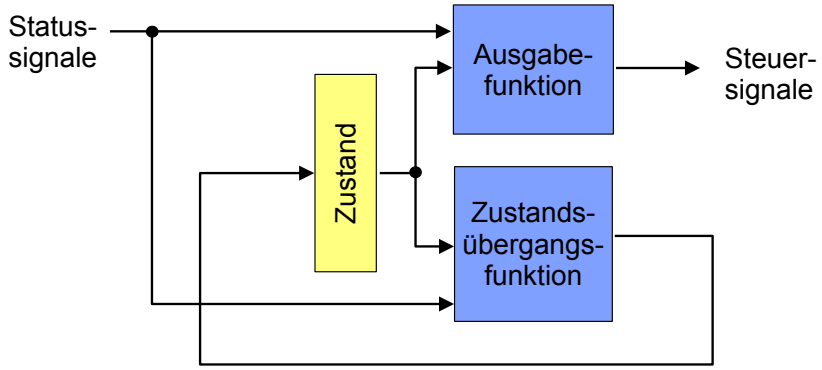
```
entity CONTROLLER is
  port (
    clk, reset,

    -- Statussignale...
    ir: in std_logic_vector (15 downto 0);
    ready, zero: in std_logic;

    -- Steuersignale...
    c_reg_ldmem, c_reg_ldi,   -- Auswahl beim Register-Laden
    c_regfile_load_lo,        -- Steuersignale Registerfile
    c_regfile_load_hi,
    c_pc_load, c_pc_inc,      -- Steuereingänge PC
    c_ir_load,                -- Steuereingang IR
    c_mem_rd, c_mem_wr,       -- Signale zum Speicher
    c_adr_pc_not_reg: out std_logic -- Auswahl Adress-Quelle
  );
end CONTROLLER;
```


b) Zustandsübergangs-Diagramm



c) Modellierung in VHDL

- **Zielstruktur:**



Hinweise

- **Falle 1:**

  Kombinatorische Zyklen und überlange Pfade in Gesamtschaltung

  => Moore-Automaten entwerfen

- **Falle 2:**

  Ungewollte Latches oder Flipflops

  => zwei Prozesse: Ausgabe- und Übergangsfunktion

  => vor case-/if-Anweisungen Default-Werte setzen

- **Weitere Hinweise**
  - Aufzählungstyp für Zustand verwenden (Lesbarkeit, bessere Optimierung bei der Synthese)
  - Ist die Belegung eines Steuersignals beliebig, sollte '-' zugewiesen werden (Optimierung)

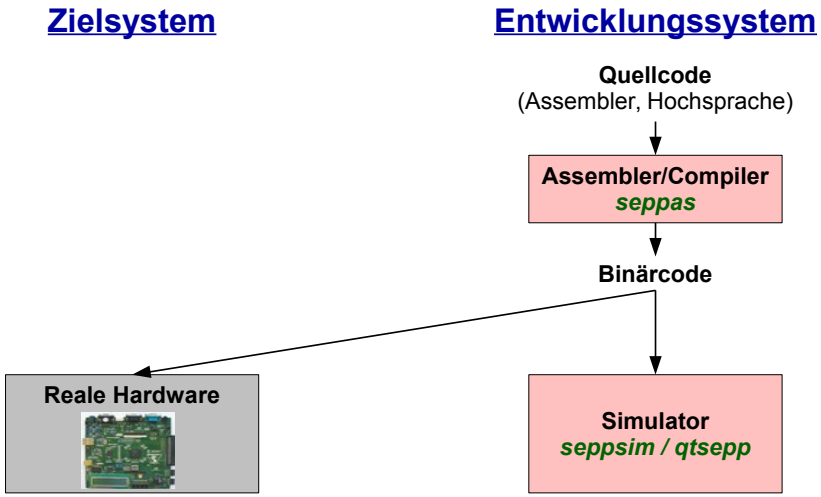
Muster

```
architecture RTL of CONTROLLER is
  -- Aufzählungstyp für den Zustand...
  type t_state is ( s_reset, s_if1, s_if2, s_id, s_alu, ... );
  signal state: t_state;
begin

  -- Prozess für die Zustandsübergangsfunktion...
  state_trans: process (clk)   -- (nur) Taktsignal in Sensitivitätsliste
  begin
    if rising_edge (clk) then
      if reset = '1' then state <= s_reset;  -- Reset hat Vorrang!
      else
        case state is
          when s_reset => state <= s_if1;
          when s_if1   => if ready = '0' then state <= s_if2; end if;
          when s_if2   => if ready = '1' then state <= s_id; end if;
          ...
          when others => null;
        end case;
      end if;
    end if;
  end process;

  ...
```


Muster

```
  ...

  -- Prozess für die Ausgabefunktion...
  output: process (state)  -- Zustand in Sensitiviätsliste
                           --(bei Mealy-Automat auch Eingänge)
  begin
    -- Default-Werte für alle Ausgangssignale...
    c_regfile_load_lo <= '0';
    c_regfile_load_hi <= '0';
    c_adr_pc_not_reg <= '-';   -- Don't Care
    ...

    -- zustandsabhängige Belegung...
    case state is
      when s_if2 =>
        c_adr_pc_not_reg <= '1';   -- hier müssen nur Abweichungen von der
        c_mem_rd <= '1';           -- Default-Belegung behandelt werden
        c_ir_load <= '1';
      when s_id =>
        c_pc_inc <= '1';
        ...
      when others => null;
    end case;
  end process;

end RTL;
```


4.4. Software-Entwicklung



SEPP

Software-**E**ntwicklungswerkzeuge
für einen **P**rozessor in **P**rogrammierbarer Logik

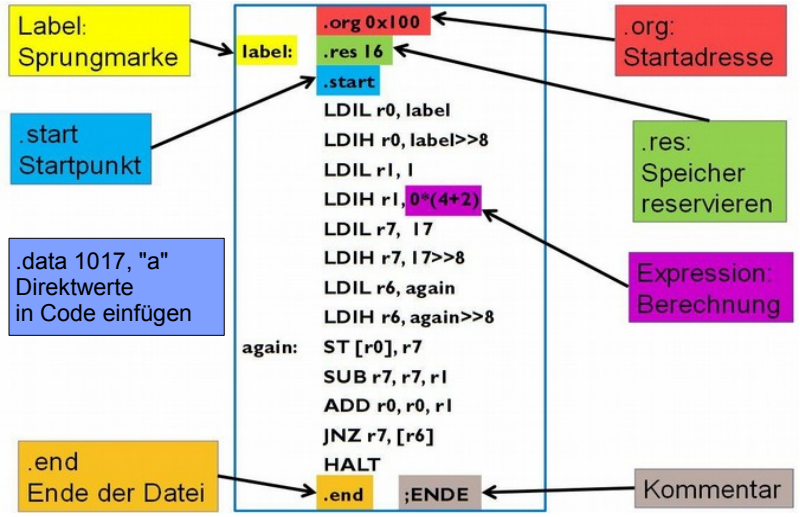
Drei Werkzeuge:

- ***SEPPas* – Assembler**
- ***SEPPsim* – Simulator**
  a) Software testen ohne verfügbare Hardware
  b) Leistungsabschätzung bei Hard- und Software (noch nicht implementiert)
- ***qtSEPP* – grafische Oberfläche**

Tool: *seppas*

- **Funktion**
  - Assembler für VISCY-Programme
- **Syntax**
  `seppas <programm>.asm`
- **Ausgabe**
  <programm>.o **:** Objekt-/Binärdatei (kein Linken mehr nötig)
- **Anmerkungen**
  - Die erzeugte Objektdatei kann sowohl in den Simulator als auch auf das Entwicklungsboard geladen werden.
  - Die Objektdatei enthält auch Debugging-Informationen (Zeilennummern, Labels) => Quell- und Binärdatei sollten gleich heißen.
  - Weitere Optionen (i.d.R. nicht benötigt): siehe Benutzerhandbuch

Beispiel einer Assembler-Datei

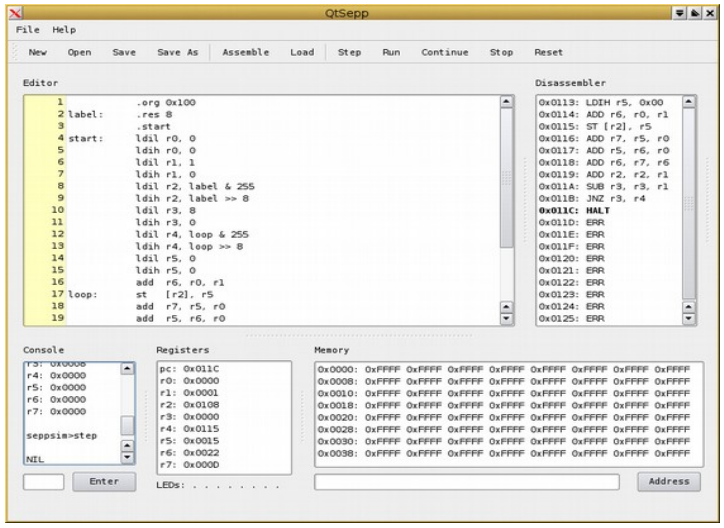


SEPPsim und *qtSEPP*

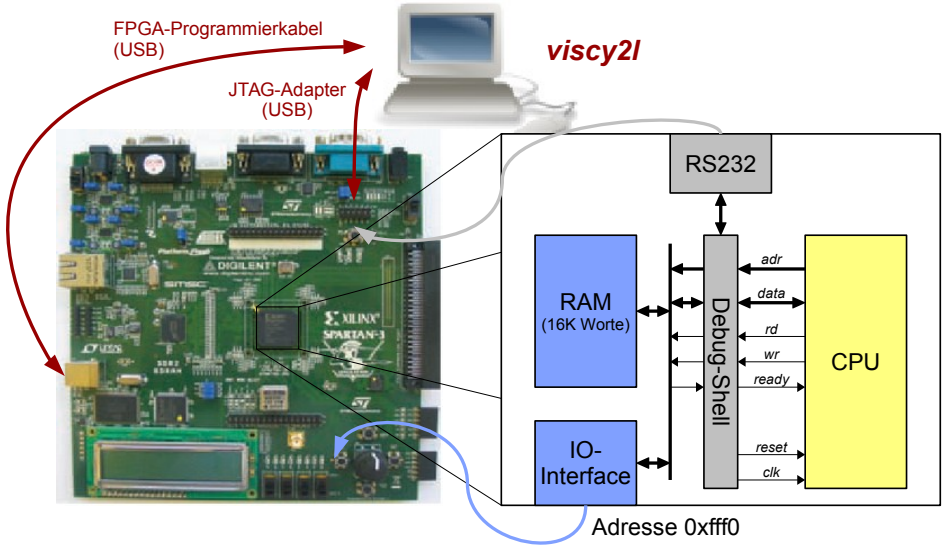
- **Frontend-/Backend-Modell**
  seppsim:
  - Backend – der eigentliche Simulator
  - ermöglicht Kommandozeilen-Bedienung und Batch-Betrieb

  qtsepp:
  - Frontend – grafische Benutzeroberfläche
- **Wichtige Funktionen in *qtsepp***
  - Compilieren (startet *seppas*)
  - "Hochladen": Objektdatei in *seppsim* laden
  - Setzen von Breakpoints, (schrittweises) Simulieren des Programmes
  - Anzeigen von:
    - Quellcode
    - Disassembler-Listing
    - Register- und Speicherinhalten

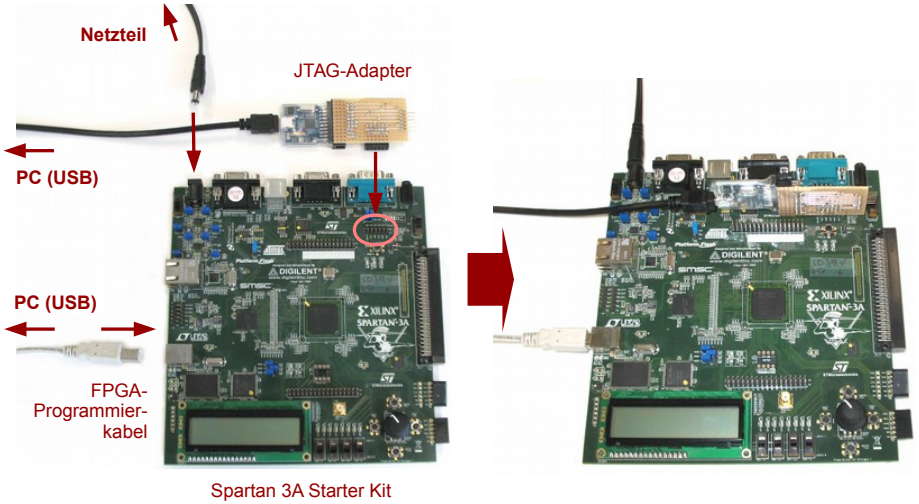
qtSEPP



4.5. Inbetriebnahme



Board anschließen



Spartan 3A Starter Kit

Ein-/Ausgabe-Schnittstelle

- **Adresse 0xfff0 schreibend**
  Bit 0-7: LEDs (0..7)

- **Adresse 0xfff0 lesend**
  Bit 0-3: Schiebeschalter
  Bit 4-7: Taster (W, S, N, O)
  Bit 8-9: Drehknopf
  Bit 10: Taster im Drehknopf

Tool: *ees-implement_viscy*

- **Funktion**
  - Einbetten einer VISCY-kompatiblen CPU in ein System mit Speicher, Debug-Schnittstelle, E/A-Anbindung
  - Platzierung, Verdrahtung & Erzeugen eines Bit-Files
- **Syntax**

  `ees-implement_viscy <cpu>.ngc`
- **Ausgabe**

  viscy.bit : komplettes System für das im Praktikum verwendete FPGA
- **Anmerkungen**
  - Die Top-Level-Entity muss "cpu" heißen, die Schnittstelle muss exakt mit der Vorgabe übereinstimmen.
  - Bei der Synthese der CPU darf die '-p'-Option nicht gesetzt sein.

Tool: *viscy2l*

- **Funktion**
  - On-Chip-Debugging des VISCY-Systems
    - CPU-Takt erzeugen (einzeln, fortlaufend)
    - Anzeigen der CPU-Pins
    - Anzeigen und Setzen von Speicherinhalten
  - Ausgabe von Objekt-Dateien in VHDL-Syntax
- **Start**

  `viscy2l [ -h ] [ -c "<Kommandos>" ]`

viscy2l – wichtige Kommandos

`help [ <Kommando> ]`	- Hilfe
`open <file name>`	- Objekt-Datei öffnen
`dump`	- Objekt-Datei ausgeben (VHDL-Syntax)
`upload`	- Objekt-Datei in (VISCY-)Speicher laden
`get <adr> [<n>]`	- (VISCY-)Speicher anzeigen
`set <adr> <val>`	- (VISCY-)Speicher schreiben
`reset`	- CPU zurücksetzen
`step`	- einzelnen CPU-Takt erzeugen & Pins anzeigen
`run_monitored`	- System taktweise ausführen & Speicherzugriffe anzeigen
`run_fullspeed`	- System autonom laufen lassen