# Teeme ise protsessori

ehk kuidas progeda **rauda**

Anti Sullin

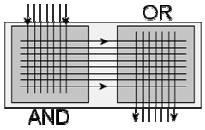
http://emperor.dyn.ee

Tevalo elektroonikahuviliste kokkutulek 2006

# Programmeeritav loogika

- PLD [programmable logic device]
- PLA [programmable logic array]
- PAL [programmable array logic])

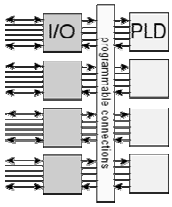
AND-OR massiiv, mõnel ka trigerid väljundis. Tavaliselt püsimäluga. Realiseerib kuni mõnisada loogikaelementi. Sobib loogikafunktsioonide realiseerimiseks, sügav loogika vajab väliseid tagasisideühendusi.




# Programmeeritav loogika (2)

- CPLD [Complex Programmable Logic Device]

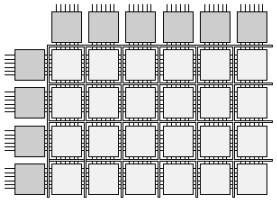
Hulk PLD sõlmesid + programmeeritavad ühendused. Tavaliselt püsimäluga (Flash). Realiseerib mõnituhat loogikaelementi. Programmeeritavad ühendused võimaldavad mõningast tagasisidet. I/O blokid võimaldavad programmeeritavaid ja/või kahesuunalisi ühendusi




# Programmeeritav loogika (3)

- FPGA [field programmable gate array]

Loogikamassiivide maatriks koos väga mitmekülgselt programmeeritavate ühendustega. Võimaldab väga sügavat loogikat ja suurel määral tagasisidestatud süsteemide realiseerimist. Tavaliselt muutmäluga. Realiseerivad kümneid tuhandeid kuni miljoneid loogikaelemente, sisaldavad lisaks veel riistvaralisi registreid, korruteid, summaatoreid jne.



# VHDL [VHSIC Hardware Description Language]

- Programmeerimiskeel
- Süntaksi poolest Ada järglane
- Sünteesitav süntaks vs simuleeritav süntaks
- Moodulite tugi
  - Lihtne koodi partitsioneerimine ja taaskasutamine
- Riistvara paralleelsuse tugi
- Avalik standard: IEEE 1076 1987/93/02

# VHDL [VHSIC Hardware Description Language]

- Erinevad kirjeldustasemed:
  - Käitumuslik
  - Registersiirete tase
  - Boole'i võrrandid
  - Loogikalülid
- Top-Down disaini metoodikate kasutamine
  - Käitumuslikul tasemel spetsifitseerimine programmina
  - Testimine ja valideerimine käitumusliku taseme suhtes koos simuleerimisega

# Sünteesitav kood

- Ajalised viited ei ole sünteesitavad
- Muutujate tüübid on piiratud
- Keeruline aritmeetika ei ole toetatud
  - Tuleb ise teha bitthaaval
- Sünteesida ei saa jadamisi täidetavat koodi

# Muutujad riistvaras

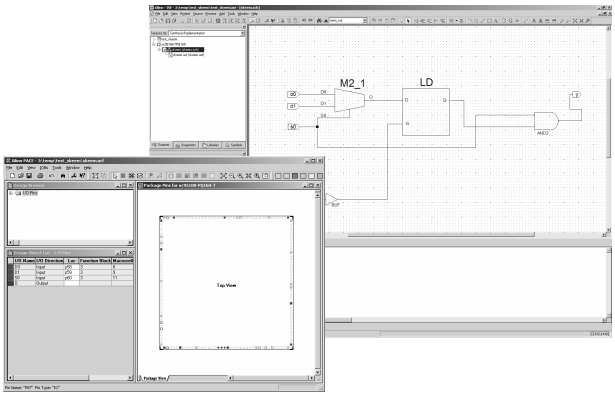
1. Muutuja == sünkroonne triger (Flip-flop):
   - kui muutujat omistatakse mingi signaali frondil
2. Muutuja == asünkroonne triger (Latch):
   - kui muutuja väärtus ei ole programmi igas harus määratud, st igas harus ei omistata
3. Muutuja == juhe
   - kui muutuja on igas harus määratud teiste muutujate poolt (st mälu ei ole vaja)

# Varjatud registrite probleem

- Kõik koodi harud peavad olema kaetud
  - ka need, kuhu kood mitte kunagi ei saa sattuda!
  - kasutades mitmevalentset loogikat, ka kõik erinevad loogikanivood!
- Lahendus: igas if / switch konstruktsioonis kasutada else haru.
- Vahetulemusi ei tohi arvutada tingimuslikus harus:
  - paralleelse loogika sünteesil ei mängi omistamiste järjekord mingit rolli!

# Esimene näide

- Riistvara süntees kasutades skeemiredaktorit.




# Teine näide

- Riistvara süntees VHDL'is

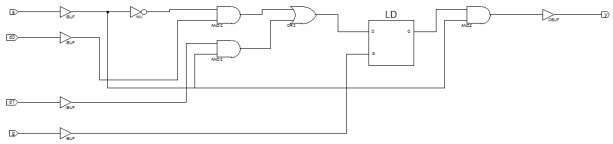
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity skeem is
    Port ( d0 : in  STD_LOGIC;
           d1 : in  STD_LOGIC;
           s : in  STD_LOGIC;
           g : in  STD_LOGIC;
           y : out  STD_LOGIC);
end skeem;
```

```
architecture RTL of skeem is
        signal mux_out : std_logic;
        signal triger_out : std_logic;
begin
        mux_out <= d0 when s = '0' else d1;

        triger: process(mux_out, g) begin
                if g = '1' then
                        triger_out <= mux_out;
                end if;
        end process triger;

        y <= s and triger_out;
end RTL;
```




# Kolmas näide

- Keerulisem riistvara – protsessorikene
  - 4-bit andmesiin
  - 4-bit i/o aadressisiin
  - 8-bit programmimälu aadressisiin
  - 16 registrit
  - 16 sõna muutmälu
  - 1 IPC
  - MOV, AND, OR, NOT, ADD, ADC, SUB, SBC, IN, OUT, LDI, LD, ST, JMP, JZ, JNZ

# Materjale

- The VHDL Cookbook http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf
- XESS Inc näidiskoodid http://www.xess.com/
- OpenCores: LGPL hardware http://www.opencores.org/

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity system is
    Port ( reset : in  STD_LOGIC;
           addrled : out  STD_LOGIC_VECTOR (6 downto 0);
           led : out  STD_LOGIC_VECTOR (6 downto 0);
         clkled: out STD_LOGIC;
           clk : in  STD_LOGIC);
end system;

architecture Behavioral of system is

signal count : STD_LOGIC_VECTOR (3 downto 0);

signal led_n: STD_LOGIC_VECTOR (6 downto 0);
signal addrled_n: STD_LOGIC_VECTOR (6 downto 0);

signal addr :STD_LOGIC_VECTOR (7 downto 0);
signal cmd :STD_LOGIC_VECTOR (11 downto 0);
signal port_id :STD_LOGIC_VECTOR (3 downto 0);
signal inputport :STD_LOGIC_VECTOR (3 downto 0);
signal outputport :STD_LOGIC_VECTOR (3 downto 0);
signal pclk :STD_LOGIC;
signal in_strobe :STD_LOGIC;
signal out_strobe :STD_LOGIC;

signal counter :STD_LOGIC_VECTOR (26 downto 0);

component mem
    port(
    A : in std_logic_vector(7 downto 0);
    SPO : out std_logic_vector(11 downto 0));
end component;

component cpu
    Port ( addr : out  STD_LOGIC_VECTOR (7 downto 0);
           cmd : in  STD_LOGIC_VECTOR (11 downto 0);
           port_id : out  STD_LOGIC_VECTOR (3 downto 0);
           inputport : in  STD_LOGIC_VECTOR (3 downto 0);
           outputport : out  STD_LOGIC_VECTOR (3 downto 0);
           clk : in  STD_LOGIC;
         reset : in STD_LOGIC;
           in_strobe : out  STD_LOGIC;
           out_strobe : out  STD_LOGIC);
end component;

begin
clkled<=pclk;
addrled<=not addrled_n;
led<=not led_n;

process (out_strobe, pclk)
begin
  if pclk'event and pclk='1' then
    if out_strobe='1' then
      count <= outputport;
    end if;
  end if;
end process;


processor:cpu
  port map(
    addr => addr,
    cmd => cmd,
    port_id => port_id,
    inputport => inputport,
    outputport => outputport,
    clk => pclk,
    reset => reset,
    in_strobe => in_strobe,
    out_strobe => out_strobe);

program:mem
  port map(
    A => addr,
    SPO => cmd);

LEDs: process(count)
  begin
    case count is
      when "0000" => led_n <= "0111111";
      when "0001" => led_n <= "0000110";
      when "0010" => led_n <= "1011011";
      when "0011" => led_n <= "1001111";
      when "0100" => led_n <= "1100110";
      when "0101" => led_n <= "1101101";
      when "0110" => led_n <= "1111101";
      when "0111" => led_n <= "0000111";
      when "1000" => led_n <= "1111111";
      when "1001" => led_n <= "1101111";
      when "1010" => led_n <= "1110111";
      when "1011" => led_n <= "1111100";
      when "1100" => led_n <= "0111001";
      when "1101" => led_n <= "1011110";
      when "1110" => led_n <= "1111001";
      when others => led_n <= "1110001";
    end case;
  end process LEDs;

ADDRLEDs: process(addr)
  begin
    case addr(3 downto 0) is
      when "0000" => addrled_n <= "0111111";
      when "0001" => addrled_n <= "0000110";
      when "0010" => addrled_n <= "1011011";
      when "0011" => addrled_n <= "1001111";
      when "0100" => addrled_n <= "1100110";
      when "0101" => addrled_n <= "1101101";
      when "0110" => addrled_n <= "1111101";
      when "0111" => addrled_n <= "0000111";
      when "1000" => addrled_n <= "1111111";
      when "1001" => addrled_n <= "1101111";
      when "1010" => addrled_n <= "1110111";
      when "1011" => addrled_n <= "1111100";
      when "1100" => addrled_n <= "0111001";
      when "1101" => addrled_n <= "1011110";
      when "1110" => addrled_n <= "1111001";
      when others => addrled_n <= "1110001";
    end case;
end process ADDRLEDs;


clock1hz: process (clk)
begin
  if clk'event and clk='1' then
    if counter<7999999 then
      counter<=counter+1;
    else
      counter<=(others=>'0');
      pclk<=not pclk;
    end if;
  end if;
end process clock1hz;

end Behavioral;
```

---

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cpu is
    Port ( addr : out  STD_LOGIC_VECTOR (7 downto 0);
           cmd : in  STD_LOGIC_VECTOR (11 downto 0);
           port_id : out  STD_LOGIC_VECTOR (3 downto 0);
           inputport : in  STD_LOGIC_VECTOR (3 downto 0);
           outputport : out  STD_LOGIC_VECTOR (3 downto 0);
           clk : in  STD_LOGIC;
         reset : in STD_LOGIC;
           in_strobe : out  STD_LOGIC;
           out_strobe : out  STD_LOGIC);
end cpu;

architecture Behavioral of cpu is

  type bus4 is array (integer range <>) of std_logic_vector(0 to
3);

  -- ALL INTERNAL WIRES & REGISTERS
  signal pc : std_logic_vector(7 downto 0);
  signal next_pc: std_logic_vector(7 downto 0);

  signal writec: std_logic;
  signal c: std_logic;
  signal writez: std_logic;
  signal z: std_logic;

  signal add_x : std_logic_vector(3 downto 0);
  signal add_y : std_logic_vector(3 downto 0);
  signal add_r : std_logic_vector(4 downto 0); -- +carry out
  signal add_c : std_logic;

  signal alu_x : std_logic_vector(3 downto 0);
  signal alu_y : std_logic_vector(3 downto 0);
  signal alu_r : std_logic_vector(3 downto 0);
  signal alu_z : std_logic;
  signal alu_c : std_logic;
  signal alu_op : std_logic_vector(2 downto 0);

  signal outdemux : std_logic_vector(15 downto 0);

  signal reg : bus4(15 downto 0);
  signal reg_in : std_logic_vector(3 downto 0);
  signal reg_out_1 : std_logic_vector(3 downto 0);
  signal reg_out_2 : std_logic_vector(3 downto 0);
  signal writereg : std_logic_vector(15 downto 0);
  signal reg_1 : std_logic_vector(3 downto 0); -- reg 1 address
  signal reg_2 : std_logic_vector(3 downto 0); -- reg 2 address
  signal reg_in_sel : std_logic_vector (1 downto 0);

  signal mem : bus4(15 downto 0);
  signal mem_in : std_logic_vector(3 downto 0);
  signal mem_out_1 : std_logic_vector(3 downto 0);
  signal mem_1 : std_logic_vector(3 downto 0); -- mem rd address
  signal writemem : std_logic_vector(15 downto 0);

  signal const : std_logic_vector(3 downto 0);

  signal reset_delay : std_logic;
  signal internal_reset : std_logic;

begin

-------------------------------------------------------------------
-- RESET LOGIC

-- internal reset f/f-s - reset delay for startup
process (clk, reset)
begin
  if reset = '1' then
    reset_delay<='1';
  else
    if clk'event and clk='1' then
      reset_delay<='0';
    end if;
  end if;
end process;

process (clk, reset)
begin
  if reset = '1' then
    internal_reset<='1';
  else
    if clk'event and clk='1' then
      if reset_delay='0' then
        internal_reset<='0';
      else
        internal_reset<='1';
      end if;
    end if;
  end if;
end process;

-------------------------------------------------------------------
-- PROGRAM COUNTER REGISTER & FLAGS REGISTER
process (clk, internal_reset)
begin
  if internal_reset = '1' then
    pc <= (others => '0');
    c <= '0';
    z <= '0';
  else
    if clk'event and clk='1' then
      pc<=next_pc;
      if writec='1' then
        c<=alu_c;
      end if;
      if writez='1' then
        z<=alu_z;
      end if;
    end if;
  end if;
end process;
addr<=pc;

-------------------------------------------------------------------
-- REGISTER FILE

-- 16 registers, 4bit each
g1: for cnt in 0 to 15 generate
  process (clk, internal_reset)
  begin
    if internal_reset = '1' then
      reg(cnt)<=(others=>'0');
    else
      if clk'event and clk='1' then
        if writereg(cnt)='1' then
          reg(cnt)<=reg_in;
        end if;
      end if;
    end if;
  end process;
end generate;


-- register file demux
-- if cmd output goes to register, this array has register's
enable signals
outdemux <=      "1000000000000000" when cmd(7 downto 4) = "1111"
else
           "0100000000000000" when cmd(7 downto 4) = "1110" else

           "0010000000000000" when cmd(7 downto 4) = "1101" else
           "0001000000000000" when cmd(7 downto 4) = "1100" else
           "0000100000000000" when cmd(7 downto 4) = "1011" else
           "0000010000000000" when cmd(7 downto 4) = "1010" else
           "0000001000000000" when cmd(7 downto 4) = "1001" else
           "0000000100000000" when cmd(7 downto 4) = "1000" else
           "0000000010000000" when cmd(7 downto 4) = "0111" else

           "0000000001000000" when cmd(7 downto 4) = "0110" else

           "0000000000100000" when cmd(7 downto 4) = "0101" else
           "0000000000010000" when cmd(7 downto 4) = "0100" else
           "0000000000001000" when cmd(7 downto 4) = "0011" else
           "0000000000000100" when cmd(7 downto 4) = "0010" else
           "0000000000000010" when cmd(7 downto 4) = "0001" else
           "0000000000000001";

-- register file outputs (multiplexers)
reg_out_1 <=     reg(0) when reg_1 = "0000" else
           reg(1) when reg_1 = "0001" else

           reg(2) when reg_1 = "0010" else
           reg(3) when reg_1 = "0011" else
           reg(4) when reg_1 = "0100" else
           reg(5) when reg_1 = "0101" else
           reg(6) when reg_1 = "0110" else
           reg(7) when reg_1 = "0111" else
           reg(8) when reg_1 = "1000" else

           reg(9) when reg_1 = "1001" else

           reg(10) when reg_1 = "1010" else
           reg(11) when reg_1 = "1011" else
           reg(12) when reg_1 = "1100" else
           reg(13) when reg_1 = "1101" else
           reg(14) when reg_1 = "1110" else
           reg(15);

reg_out_2 <=     reg(0) when reg_2 = "0000" else
           reg(1) when reg_2 = "0001" else

           reg(2) when reg_2 = "0010" else
           reg(3) when reg_2 = "0011" else
           reg(4) when reg_2 = "0100" else
           reg(5) when reg_2 = "0101" else
           reg(6) when reg_2 = "0110" else
           reg(7) when reg_2 = "0111" else
           reg(8) when reg_2 = "1000" else

           reg(9) when reg_2 = "1001" else

           reg(10) when reg_2 = "1010" else
           reg(11) when reg_2 = "1011" else
           reg(12) when reg_2 = "1100" else
           reg(13) when reg_2 = "1101" else
           reg(14) when reg_2 = "1110" else
           reg(15);

-- REG FILE IN MUX
-- data can be stored to register file from 4 sources depending
the command
process (alu_r, mem_out_1, inputport, const, reg_in_sel)
begin
  case reg_in_sel is
    when "01" => reg_in <= mem_out_1;      -- reg <= mem
    when "10" => reg_in <= inputport;      -- reg <= input port
    when "11" => reg_in <= const;          -- reg <= constant
value in command
    when others => reg_in <= alu_r;        -- reg <= alu result
  end case;
end process;

-------------------------------------------------------------------
-- RAM

-- RAM registers: 16 words, 4 bits each
process (clk, internal_reset)
variable cnt : integer range 0 to 15;
begin
  if internal_reset='1' then
    for cnt in 0 to 15 loop
      mem(cnt)<=(others=>'0');
    end loop;
  else
    if clk'event and clk='1' then
      for cnt in 0 to 15 loop
        if writemem(cnt)='1' then
          mem(cnt)<=mem_in;
        end if;
      end loop;
    end if;
```

```
  end if;
end process;

-- RAM output demux
mem_out_1 <=   mem(0) when mem_1 = "0000" else
           mem(1) when mem_1 = "0001" else

           mem(2) when mem_1 = "0010" else
           mem(3) when mem_1 = "0011" else
           mem(4) when mem_1 = "0100" else
           mem(5) when mem_1 = "0101" else
           mem(6) when mem_1 = "0110" else
           mem(7) when mem_1 = "0111" else
           mem(8) when mem_1 = "1000" else

           mem(9) when mem_1 = "1001" else

           mem(10) when mem_1 = "1010" else
           mem(11) when mem_1 = "1011" else
           mem(12) when mem_1 = "1100" else
           mem(13) when mem_1 = "1101" else
           mem(14) when mem_1 = "1110" else
           mem(15);

mem_in<=reg_out_1;

-----------------------------------------------------------------
-- ALU - ARITHMETIC AND LOGIC UNIT

-- ADDER / SUBTRACTER
-- one adder is here, its input and output is switched below for
multiple functions
process (add_x, add_y, add_c)
begin
  -- result has one extra bit - carry out
  add_r <= ("0"&add_x) + ("0"&add_y) + ("0000"&add_c);
end process;

-- ALU CORE
process (alu_x, alu_y, c, alu_op, add_r)
begin
  alu_c <= '0'; --default carry flag

  add_x <= alu_x; -- default adder connections
  add_y <= alu_y; -- everything must be always connected
somewhere
  add_c <= c;

  -- actual alu result is multiplexed from different
  -- functional blocks depending on the command given
  case alu_op is
    when "000" => alu_r <= alu_y;                    -- mov: r <=
y
    when "001" => alu_r <= alu_x and alu_y;          -- and: r <=
x and y
    when "010" => alu_r <= alu_x or alu_y;           -- or: r<= x
or y
    when "011" => alu_r <= not alu_y;                        --
not: r<=not y
    when "100" =>                                    -- add:
(r,c) <= x + y
      add_c <= '0';
      alu_r <= add_r(3 downto 0);
      alu_c <= add_r(4);
    when "101" =>                                    -- addc:
(r,c) <= x + y + c
      alu_r <= add_r(3 downto 0);
      alu_c <= add_r(4);
    when "110" =>                                    -- sub:
(r,c) <= x - y
      add_c <= '1';
      add_y <= not alu_y;
      alu_r <= add_r(3 downto 0);
      alu_c <= add_r(4);
    when others =>                                   -- subc:
(r,c) <= x - y - c
      add_c <= not c;
      add_y <= not alu_y;
      alu_r <= add_r(3 downto 0);
      alu_c <= add_r(4);
  end case;
  alu_z <= alu_r(3) or alu_r(2) or alu_r(1) or alu_r(0); -- next
zero flag
end process;

-- ALU input is hardwired to register file output at the moment
alu_x <= reg_out_1;
alu_y <= reg_out_2;

-----------------------------------------------------------------
-- PROCESSOR CORE - COMMAND DECODER UNIT
process (cmd, pc, z, outdemux)
begin
  -- default values (can be changed later, nothing can be left
unconnected)
  writereg <= (others => '0');    -- no bits are latched to
register
  writemem <= (others => '0');    -- no bits are latched to memory
  next_pc <= pc + 1;                               -- next
command address
  in_strobe <= '0';
  out_strobe <= '0';
  reg_in_sel <= "00";                      -- register file input
port is connected to alu output
  writec <= '0';                           -- disable latching
flags on next clock
  writez <= '0';

  if cmd(11) ='0' then
    writereg <= outdemux;  -- alu op - every command writes to
register file register selected by outdemux (=x)
    writec <= '1';                                   -- every
command updates flags
    writez <= '1';
  else
    case cmd(10 downto 8) is      -- not alu command
      when "000" => -- read data from in port
        in_strobe <= '1'; -- indicate external hardware that read
occurs on next clk
        reg_in_sel <= "10"; -- switch register file input to
external input
        writereg <= outdemux;
      when "001" => -- write data to out port
        out_strobe <= '1';        -- indicate external hardware to
read data from output port on next clk
      when "010" => -- load immediate
        reg_in_sel <= "11";       -- switch reg file input to
immediate operand from command code
        writereg <= outdemux;
      when "011" => -- load mem
        reg_in_sel <= "01";       -- switch reg file input to
memory output
        writereg <= outdemux;
      when "100" => -- store mem
        writemem <= outdemux; -- enable latching data at given
memory address
      when "101" => -- jmp
        next_pc <= cmd(7 downto 0);
      when "110" => -- jmp zero
        if z='1' then -- conditional jump
          next_pc <= cmd(7 downto 0);
        end if;
      when others => -- (111) jmp not zero
        if z='0' then -- conditional jump
          next_pc <= cmd(7 downto 0);
        end if;
    end case;
  end if;
end process;

-- other direct connections
const <= cmd(3 downto 0);

outputport<=reg_out_1;
port_id <= cmd(7 downto 4);

reg_1 <= cmd(7 downto 4);
reg_2 <= cmd(3 downto 0);

mem_1 <= cmd(3 downto 0);

alu_op <= cmd(10 downto 8);

end Behavioral;
```

---

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mem IS
  port (
  a: IN std_logic_VECTOR(7 downto 0);
  spo: OUT std_logic_VECTOR(11 downto 0));
END mem;

ARCHITECTURE mem_a OF mem IS
begin

cntr: process(a)
begin
  case a is
    when "00000000" => spo <= "101000010001";
    when "00000001" => spo <= "101000000000";
    when "00000010" => spo <= "010000000001";
    when "00000011" => spo <= "100100000000";
    when others => spo <= "110100000010";
  end case;
end process cntr;

END mem_a;
```