UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	**INF3430 Digital systemkonstruksjon**
Eksamensdag:	**9. desember 2013**
Tid for eksamen:	**9-13**
Oppgavesettet er på 12 sider	
Vedlegg:	**1**
Tillatte hjelpemidler:	**Alle trykte og skrevne hjelpemidler tillatt, samt kalkulator.**

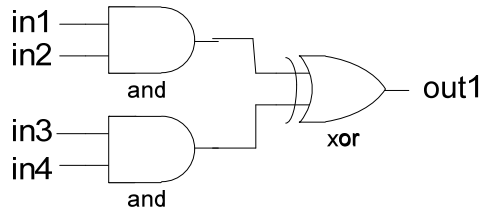
Kontroller at oppgavesettet er komplett
før du begynner å besvare spørsmålene.

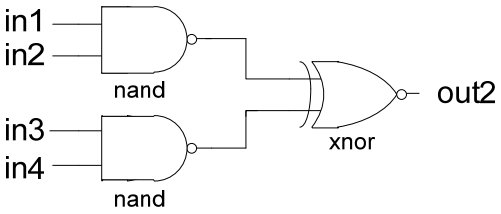
Oppgaveteksten består av oppgave 1–4 (flervalgsoppgaver) som skal besvares på skjemaet som er vedlagt etter oppgaveteksten og oppgave 5-10 som besvares på vanlige ark. Oppgave 1-4 har til sammen vekt på 20%, mens oppgave 5-9 har til sammen vekt på 30% og oppgave 10 har til sammen vekt på 50%.

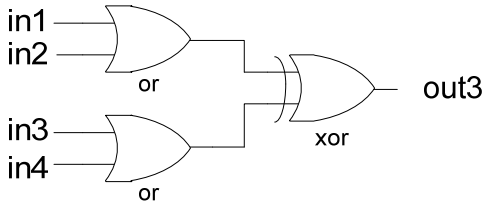
Generelt for oppgave 1-4:
Hver oppgave består av et tema i venstre kolonne og en del utsagn hver angitt med en stor bokstav. Oppgavene besvares ved å merke tydelige kryss (X) i rett kolonne for riktig svaralternativ (dvs. at et utsagn er sant) i skjemaet i vedlegget. Det er alltid *minst en* riktig avmerking for hver oppgave, men det er ofte *flere* riktige avmerkninger. *For å få best karakter skal man sette flere kryss i en oppgave hvis det er flere riktige utsagn.* Det gis 1 poeng for hver avkrysning der det skal være avkrysning. Det gis -1 poeng for hver avkrysning der det ikke skal være avkrysning. Mangel på kryss der det skal være kryss gir også -1 poeng. Du kan benytte høyre kolonne i oppgaveteksten til kladd. Skjema påført ditt kandidatnummer i vedlegget er din besvarelse.

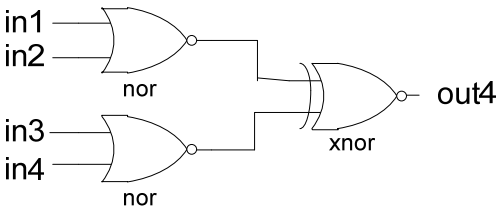
Oppgave 1 (vekt 5%)

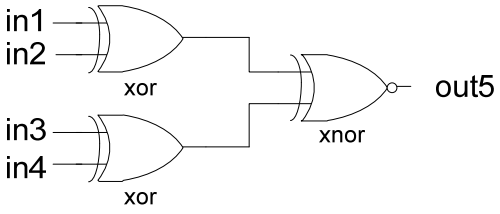
Figuren under viser de kombinatoriske kretsene med out1 lik “and-and-xor”, out2 lik “nand-nand-xnor”, out3 lik “or-or-xor”, out4 lik “nor-nor-xnor” og out5 lik “xor-xor-xnor”.











En 4-input Xilinx Spartan-3 LUT med innhold **”7888”** (hex) realiserer en:	A	and-and-xor	
	B	nand-nand-xnor	
	C	or-or-xor	
	D	nor-nor-xnor	
	E	xor-xor-xnor	

Oppgave 2 (vekt 5%)

Kan variabler i VHDL deklareres i:	A	Prosess	
	B	Architecture	
	C	Procedure	
	D	Function	
	E	Entity	

Oppgave 3 (vekt 5%)

Konstruksjon 1	A	Xilinx har harde DSP moduler som har MAC (Multiply and Accumulate) funksjon.	
	B	Block RAM (BRAM) i FPGA har ikke en kjent initialverdi etter konfigurering.	
	C	Det blir ikke mindre tilgjengelig logikk i FPGA'en ved bruk av dedikert mentelogikk i FPGA'en.	
	D	Ved synkron styring av flip-flop'er med set/reset er det best at set kommer før reset i VHDL koden.	
	E	I FPGA-teknologi har forbindelseslinjer mellom LUT'er vanligvis mindre tidsforsinkelse enn gjennom LUT'er.	

Oppgave 4 (vekt 5%)

Konstruksjon 2	A	Det er raskere å simulere med variabler enn med signaler i VHDL.	
	B	For hver delta cycle i en VHDL simulator går tiden 1 nanosekund.	
	C	Selv om antall input til en funksjon er konstant øker forbruket av LUT'er med kompleksiteten til funksjonen.	
	D	En Xilinx Block RAM (BRAM) har to porter som kan leses fra og skrives til i samme klokkeperiode.	
	E	Ved delvis rekonfigurasjon vil de delene av kretsen som ikke blir rekonfigurert også være inaktive under rekonfigurering.	

Oppgave 5 (for INF3430) (vekt 6%)

Under er det oppgitt VHDL entiteten til modulen *strobegen*.

Modulen klokkes med klokkesignalet *mclk* og resettes med det asynkrone og aktivt høye resetsignalet *rst*. Modulen mottar data på inngangen *data*. Det asynkrone inngangssignalet *data* må synkroniseres med klokken *mclk* før det brukes i modulen for å unngå metastabilitet.

Modulen skal generere *rising_str* signalet som er aktivt en klokkeperiode når *data* endres fra '0' til '1', og *falling_str*-signalet som er aktivt en klokkeperiode når data endres fra '1' til '0'. I tillegg skal *cycle_cnt* vise antall klokkeperioder mellom signalene *rising_str* blir '1' og *falling_str blir '1'* og settes ut når *falling_str*-signalet blir aktivt høyt. *cycle_cnt kan ha* maksverdi x"FFFF". Signalet *cycle_cnt* skal beholde verdien til neste gang *falling_str* blir aktivt høyt. Når *rst* er aktivt høyt skal *rising_str* og *falling_str* settes til '0', og *cycle_cnt* settes lik null.

Implementer VHDL arkitekturen rtl til entiteten *strobegen*.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity strobegen is
  rst         : in  std_logic;
  mclk        : in  std_logic;
  data        : in  std_logic;
  rising_str  : out std_logic;
  falling_str : out std_logic;
  cycle_cnt   : out std_logic_vector(15 downto 0));
end strobegen;

architecture rtl of strobegen is

  < Skriv VHDL kode her >

end rtl;
```

Oppgave 6 (vekt 6%)

I modulen *compute* som er vist under, utføres først en sammenligning b > c og deretter utføres beregningen av utgangsverdien *result* avhengig av resultatet av sammenligningen.

Det viser seg at VHDL koden ikke overholder timingkravet til FPGA-kretsen som skal brukes. Forsinkelsen gjennom modulen må nesten halveres for å overholde timingkravet. Dette kan løses ved at VHDLkoden endres til at det legges til en klokkeperiode med pipelining slik at utgangssignalet *result* kommer en klokkeperiode senere enn i arkitekturen vist under.

Lag en ny arkitektur *rtl2* til modulen *compute* med samme funksjon, men med en ekstra klokkeperiode pipelining som løser timingproblemet.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compute is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    a         : in  unsigned(3 downto 0);
    b         : in  unsigned(3 downto 0);
    c         : in  unsigned(3 downto 0);
    result    : out unsigned(3 downto 0));
end compute;

architecture rtl of compute is
begin

  process (rst, mclk) is
  begin
    if rst='1' then
      result <= (others => '0');
    elsif rising_edge(mclk) then
      if b > c then
        result <= a + b;
      else
        result <= a + c;
      end if;
    end if;
  end process;

end rtl;
```

Oppgave 7 (vekt 6%)

Modulen *rstmodule* som er vist under klokkes med klokkesignalet *mclk* og resettes med det asynchrone og aktivt høye resetsignalet *arst*.

Forklar **kort med ord** funksjonaliteten til prosessen P_RST og hvorfor componenten *bufg* inngår i modulen.

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.all;

entity rstmodule is
  port (
    arst        : in  std_logic;
    mclk        : in  std_logic;
    rst         : out std_logic);
end rstmodule;

architecture rtl of rstmodule is
  signal rst_s1, rst_s2 : std_logic;

  component bufg
    port(i : in std_logic;
         o : out std_logic);
  end component;

begin

  P_RST : process(arst, mclk) is
  begin
    if arst='1' then
      rst_s1 <= '1';
      rst_s2 <= '1';
    elsif rising_edge(mclk) then
      rst_s1 <= '0';
      rst_s2 <= rst_s1;
    end if;
  end process P_RST;

  bufg_inst: bufg
    port map (i => rst_s2,
              o => rst);

end rtl;
```

Oppgave 8 (vekt 6%)

I VHDL koden som er vist under, er det 5 feil/mangler.

Finn de 5 feilene/manglene og oppgi endringene i VHDL koden så feilene/manglene fjernes.

```
library ieee;
use ieee.std_logic_1164.all;

entity faulty is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    enable    : in  std_logic;
    data      : in  std_logic_vector(7 downto 0);
    equals    : out std_logic;
    term_cnt  : out std_logic);
end faulty;

architecture rtl of faulty is
  signal count: std_logic_vector(7 downto 0);
begin
  P_COMPARE: process(data) is
  begin
    if data=count then
      equals <= '1';
    else
      equals <= '0';
    end if;
  end process;

  P_COUNT: process(rst, mclk) is
  begin
    if rising_edge(mclk) then
      count <= count + 1;
    end if;
  end process;

  term_cnt <= 'Z' when enable='0' else
              '1' when count="1111" else
              '0';
end rtl;
```

Oppgave 9 (vekt 6%)

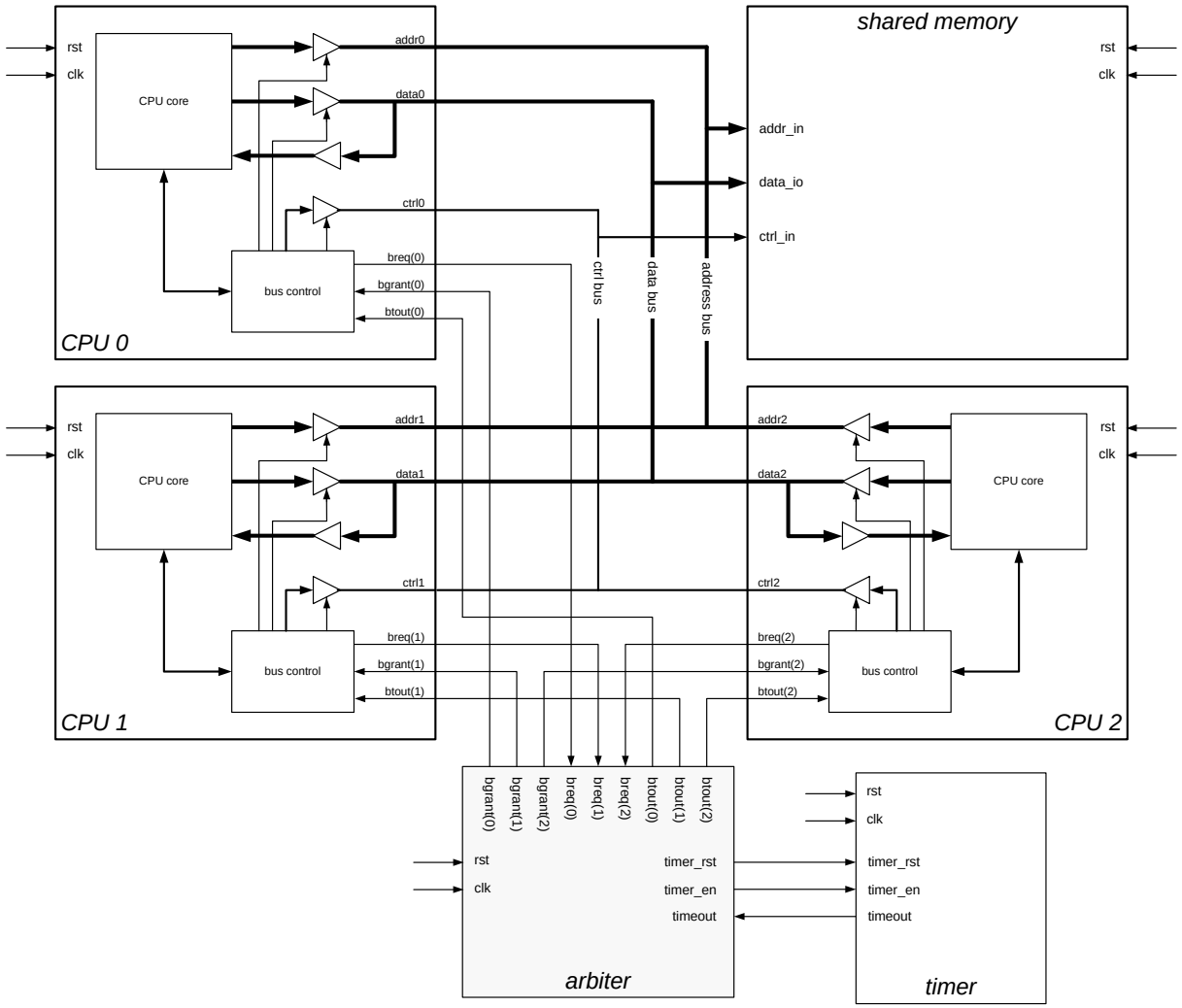
Under er det oppgitt entiteten og arkitekturen til modulen *something*.
Beskriv **kort med ord** funksjonaliteten til modulen *something*, og tegn et timingdiagram med alle signalene oppgitt i entiteten med påtrykksverdier til signalene *rst*, *mclk*, *send_str* og *din* de første 20 klokkeperiodene etter at reset blir inaktiv lav ('0') med signalet *din* med verdien x"13" slik at funksjonaliteten vises.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity something is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    send_str  : in  std_logic;
    din       : in  std_logic_vector(7 downto 0);
    dout      : out std_logic;
    sclk      : out std_logic;
    cs_n      : out std_logic);
end something;

architecture rtl of something is
  type state_type is (IDLE, START, PHASE0, PHASE1);
  signal present_state, next_state  : state_type;
  signal dreg, next_dreg      : std_logic_vector(7 downto 0);
  signal cnt, next_cnt        : unsigned(4 downto 0);
  signal next_dout            : std_logic;
  signal next_sclk            : std_logic;
  signal next_cs_n            : std_logic;
  signal dout_i               : std_logic;
  signal sclk_i               : std_logic;
  signal cs_i_n               : std_logic;
begin

  P_SEQ: process (rst, mclk) is
  begin
    if (rst = '1') then
      dout_i        <= '0';
      sclk_i        <= '1';
      cs_i_n        <= '1';
      dreg          <= (others => '0');
      cnt           <= (others => '0');
      present_state <= IDLE;
    elsif rising_edge(mclk) then
      dout_i        <= next_dout;
      sclk_i        <= next_sclk;
      cs_i_n        <= next_cs_n;
      dreg          <= next_dreg;
      cnt           <= next_cnt;
      present_state <= next_state;
    end if;
  end process P_SEQ;
```

```
P_COMB: process (present_state, send_str, din, cnt, dreg,
                 dout_i, sclk_i, cs_i_n) is
  begin
    next_dout   <= dout_i;
    next_sclk   <= sclk_i;
    next_cs_n   <= cs_i_n;
    next_dreg   <= dreg;
    next_cnt    <= cnt;
    next_state  <= present_state;

    case present_state is
      when IDLE =>
        next_sclk   <= '1';
        next_cs_n   <= '1';
        next_dout   <= '0';
        next_dreg   <= (others => '0');
        next_cnt    <= (others => '0');
        if send_str='1' then
          next_dreg   <= din;
          next_state  <= START;
        end if;
      when START =>
        next_cnt <= cnt + 1;
        if cnt(0)='1' then
          next_cs_n   <= '0';
          next_dout   <= dreg(7);
          next_dreg   <= dreg(6 downto 0) & '0';
          next_state  <= PHASE0;
        end if;
      when PHASE0 =>
        next_cnt <= cnt + 1;
        if cnt(0)='0' then
          next_sclk   <= '0';
          next_state  <= PHASE1;
        end if;
      when others =>
        next_cnt <= cnt + 1;
        if (cnt(4)='1' and cnt(0)='1') then
          next_sclk   <= '1';
          next_cs_n   <= '1';
          next_dout   <= '0';
          next_state <= IDLE;
        elsif cnt(0)='1' then
          next_sclk   <= '1';
          next_dout   <= dreg(7);
          next_dreg   <= dreg(6 downto 0) & '0';
          next_state  <= PHASE0;
        end if;
    end case;
  end process P_COMB;

  dout   <= dout_i;
  sclk   <= sclk_i;
  cs_n   <= cs_i_n;

end rtl;
```

Oppgave 10 (vekt 50%)

Vi skal i denne oppgaven se på en *bus arbiter*, norsk: *bus-arbitrerer***.**



Figur 1. Et multiprocessorsystem med 3 CPU*er og delt minne*, shared memory

I et multiprosessorsystem kan det være to eller flere mikroprosessorer, *CPU*er, som deler et felles minne, *shared memory*..
De aksesserer dette gjennom et bussystem som består av *address, data og ctrl bus (control bus).* De deler dermed alle disse signalene. Det er meget viktig at bare en enhet av gangen sender data ut på bussen. Derfor er bussignalene fra de enkelte *CPU*ene skilt fra bussen med tristate-buffere, se figuren over.

En *bus arbiter* er en enhet som til enhver tid har kontroll på hvilke enhet som skal ha tilgang til bussen. Dette styres gjennom en prosess som kalles "*bus arbitration*", norsk: *bus-arbitrering.*
Hver *CPU i, i=0,1,2* har to arbitreringssignaler: *breq(i) og bgrant(i)*[1]**.**
Når en *CPU* ønsker tilgang til bussen setter den *breq(i)*-signalet aktivt høyt ('1'). *bus-arbitrereren* svarer ved å sette *bgrant(i)*-signalet aktivt høyt ('1'). Det er ikke sikkert dette

[1] breq = bus request, bgrant = bus grant, btout = bus timeout

skjer med en gang fordi det kan være andre *CPU*er som ikke er ferdig med å benytte bussen og det kan være andre *CPU*er som også ønsker å benytte bussen, men har høyere prioritet. Når *bgrant(i)*-signalet går aktivt forteller det *CPU*en at den kan åpne sine tristate-buffere for å aksessere bussen. Når *CPU*en er ferdig med en overføring settes *breq(i)* -signalet inaktivt lavt ('0') og sørger med dette for at andre *CPU*er kan slippe til.

For å dele bussen rettferdig er det vanlig å endre prioriteten til de enkelt *breq(i)*-signalene dynamisk. Det betyr for eksempel at hvis vi har følgende synkende prioritet: *breq(0)> breq(1)> breq(2)* så vil *breq(0)* få aksess først uansett om *breq(1)* og/eller *breq(2)* er aktive samtidig. Etter at *breq(0)* settes inaktivt og aktivt igjen straks etterpå så har *breq(0)* fått laveste prioritet og kan ikke få tilgang til bussen før både *breq(1)* og *breq(2)* er inaktive.

Dette betyr at den som sist har hatt tilgang til bussen vil få lavest prioritet neste gang to eller flere *breq*-signaler er aktive samtidig. I vårt system kan vi kan ha følgende prioritetsendringer: *breq(0)>*[2]*breq(1)>breq(2)* endres til *breq(1)>breq(2)>breq(0)* endres til *breq(2)>breq(0)>breq(1)* som igjen endres til *breq(0)>breq(1)>breq(2)* osv…

I figuren under vises prioriteringen mellom *breq(0), breq(1)* og *breq(2)* når alle *breq*-signalene settes aktivt høye ('1') samtidig. Deretter settes *breq(i)* aktivt høyt igjen med en gang *bgrant(i)* har blitt inaktivt. Dermed vises prioriteringen mellom *breq(i)*-signalene.

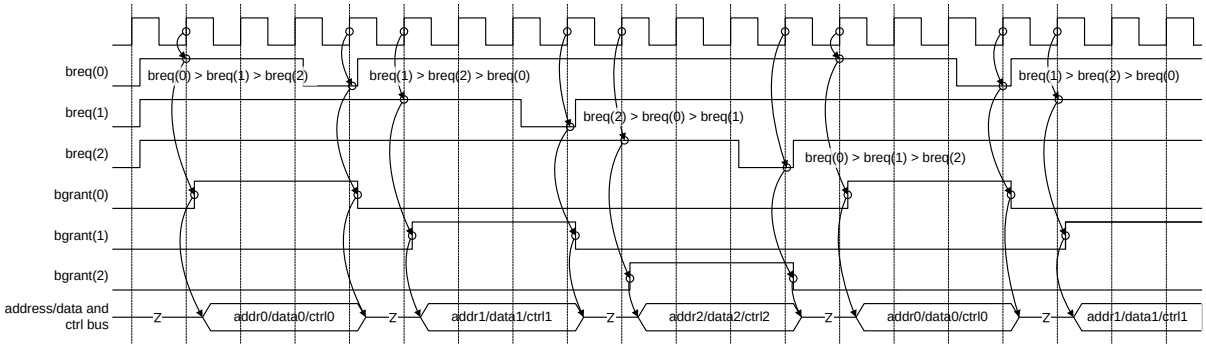


Figure 2. Bus-request og prioritering

I tillegg skal man sørge for at hver *CPU* ikke kan bruke for lang tid på bussen. Dette gjøres ved å aktivere en timer med *timer_en* (aktivt høyt) i tilstanden som *bgrant(i)* er aktivt. Dersom *timeout* går aktivt (høyt) så har man en timeoutsituasjon som flagges med at signalet *btout(i)* går aktivt høyt *og* lagres i et register, samtidig med at *bgrant(i)* settes inaktivt lavt. Når *btout(i)* går aktivt skal *CPU i* svare ved å sette *breq(i)* inaktivt lavt i neste klokkeperiode; se figur 3. *btout(i)* resettes neste gang *breq(i)* har gått aktivt og *CPU i* har fått tilgang til bussen (norsk-engelsk:"*grantet*" bussen). *timer*-modulen er vist i figur 1.

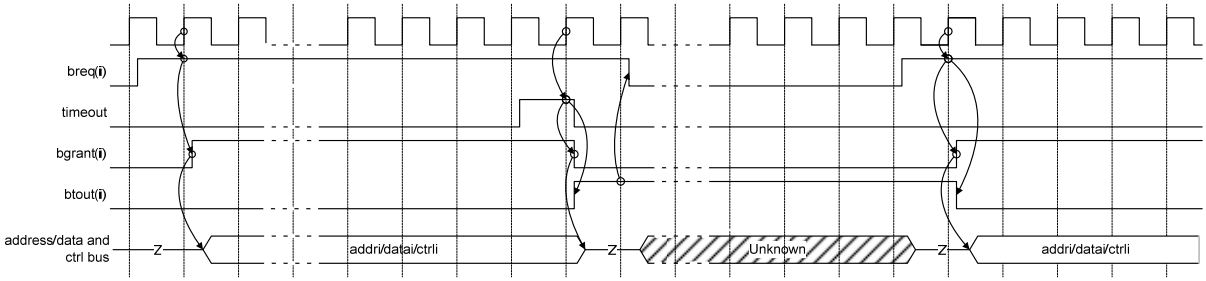


Figure3. Bus-request med timeout

[2] ">" betyr i denne sammenhengen større prioritet enn

a) Vekt 15%
Lag et ASM-flytdiagrammet som beskriver en tilstandsmaskin, *arbiter*, som arbitrerer mellom de tre *CPU*ene i figuren over og med *timeout* -funksjonen. Tilstandsmaskinen skal beskrives som en Moore-maskin**.**

Benytt følgende entitet:

```
entity arbiter is
  port
    (
      rst       : in  std_logic;
      clk       : in  std_logic;
      breq      : in  std_logic_vector(2 downto 0);
      bgrant    : out std_logic_vector(2 downto 0);
      btout     : out std_logic_vector(2 downto 0);
      timer_rst : out std_logic;
      timer_en  : out std_logic;
      timeout   : in  std_logic
      );
end entity arbiter;
```

b) Vekt 15%
Implementer tilstandsmaskinen i a) som en to-process VHDL-beskrivelse.

c) Vekt 15%
Lag en testbenk som tester tilstandsmaskinen beskrevet i b).

d) Vekt 5%
Gjøre rede for prinsippene for en selvsjekkende (selvtestende) testbenk. Beskriv kort med ord hva som skal til for å gjøre testbenken i punkt c) selvsjekkende.

INF3430. Oppgavesvar for kandidat nr: _____

Oppgave	A	B	C	D	E
1					
2					
3					
4					

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	**INF4431 Digital systemkonstruksjon**
Eksamensdag:	**9. desember 2013**
Tid for eksamen:	**9-13**
Oppgavesettet er på 12 sider	
Vedlegg:	**1**
Tillatte hjelpemidler:	**Alle trykte og skrevne hjelpemidler tillatt, samt kalkulator.**

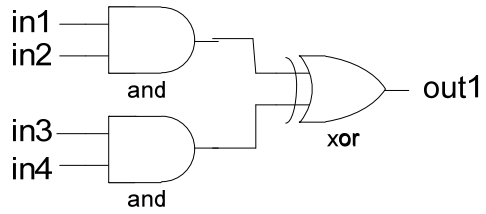
Kontroller at oppgavesettet er komplett
før du begynner å besvare spørsmålene.

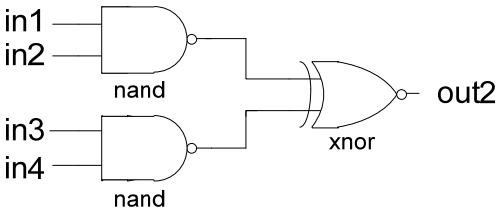
Oppgaveteksten består av oppgave 1–4 (flervalgsoppgaver) som skal besvares på skjemaet som er vedlagt etter oppgaveteksten og oppgave 5-10 som besvares på vanlige ark. Oppgave 1-4 har til sammen vekt på 20%, mens oppgave 5-9 har til sammen vekt på 30% og oppgave 10 har til sammen vekt på 50%.

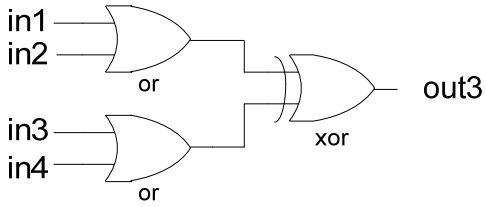
Generelt for oppgave 1-4:
Hver oppgave består av et tema i venstre kolonne og en del utsagn hver angitt med en stor bokstav. Oppgavene besvares ved å merke tydelige kryss (X) i rett kolonne for riktig svaralternativ (dvs. at et utsagn er sant) i skjemaet i vedlegget. Det er alltid *minst en* riktig avmerking for hver oppgave, men det er ofte *flere* riktige avmerkninger. *For å få best karakter skal man sette flere kryss i en oppgave hvis det er flere riktige utsagn.* Det gis 1 poeng for hver avkrysning der det skal være avkrysning. Det gis -1 poeng for hver avkrysning der det ikke skal være avkrysning. Mangel på kryss der det skal være kryss gir også -1 poeng. Du kan benytte høyre kolonne i oppgaveteksten til kladd. Skjema påført ditt kandidatnummer i vedlegget er din besvarelse.

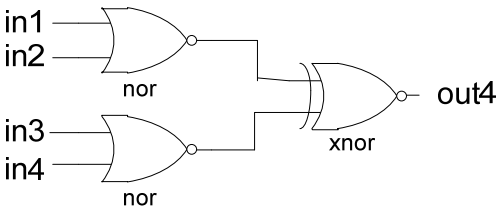
Oppgave 1 (vekt 5%)

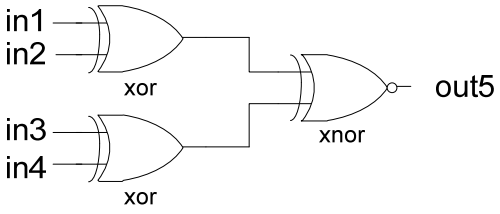
Figuren under viser de kombinatoriske kretsene med out1 lik “and-and-xor”, out2 lik “nand-nand-xnor”, out3 lik “or-or-xor”, out4 lik “nor-nor-xnor” og out5 lik “xor-xor-xnor”.











En 4-input Xilinx Spartan-3 LUT med innhold **”7888”** (hex) realiserer en:	A	and-and-xor	
	B	nand-nand-xnor	
	C	or-or-xor	
	D	nor-nor-xnor	
	E	xor-xor-xnor	

Oppgave 2 (vekt 5%)

Kan variabler i VHDL deklareres i:	A	Prosess	
	B	Architecture	
	C	Procedure	
	D	Function	
	E	Entity	

Oppgave 3 (vekt 5%)

Konstruksjon 1	A	Xilinx har harde DSP moduler som har MAC (Multiply and Accumulate) funksjon.	
	B	Block RAM (BRAM) i FPGA har ikke en kjent initialverdi etter konfigurering.	
	C	Det blir ikke mindre tilgjengelig logikk i FPGA'en ved bruk av dedikert mentelogikk i FPGA'en.	
	D	Ved synkron styring av flip-flop'er med set/reset er det best at set kommer før reset i VHDL koden.	
	E	I FPGA-teknologi har forbindelseslinjer mellom LUT'er vanligvis mindre tidsforsinkelse enn gjennom LUT'er.	

Oppgave 4 (vekt 5%)

Konstruksjon 2	A	Det er raskere å simulere med variabler enn med signaler i VHDL.	
	B	For hver delta cycle i en VHDL simulator går tiden 1 nanosekund.	
	C	Selv om antall input til en funksjon er konstant øker forbruket av LUT'er med kompleksiteten til funksjonen.	
	D	En Xilinx Block RAM (BRAM) har to porter som kan leses fra og skrives til i samme klokkeperiode.	
	E	Ved delvis rekonfigurasjon vil de delene av kretsen som ikke blir rekonfigurert også være inaktive under rekonfigurering.	

Oppgave 5 (for INF4431) (vekt 6%)

Under er det oppgitt SystemVerilog modulen *strobegen* sine interface signaler.

Modulen klokkes med klokkesignalet *mclk* og resettes med det asynkrone og aktivt høye resetsignalet *rst*. Modulen mottar data på inngangen *data*. Det asynkrone inngangssignalet *data* må synkroniseres med klokken *mclk* før det brukes i modulen for å unngå metastabilitet.

Modulen skal generere *rising_str* signalet som er aktivt en klokkeperiode når *data* endres fra '0' til '1', og *falling_str*-signalet som er aktivt en klokkeperiode når data endres fra '1' til '0'. I tillegg skal *cycle_cnt* vise antall klokkeperioder mellom signalene *rising_str* blir '1' og *falling_str blir '1'* og settes ut når *falling_str*-signalet blir aktivt høyt. *cycle_cnt kan ha* maksverdi x"FFFF". Signalet *cycle_cnt* skal beholde verdien til neste gang *falling_str* blir aktivt høyt. Når *rst* er aktivt høyt skal *rising_str* og *falling_str* settes til '0', og *cycle_cnt* settes lik null.

Implementer SystemVerilog modulen *strobegen*.

```
module strobegen(output logic rising_str,
                 output logic falling_str,
                 output logic [15:0] cycle_cnt,
                 input  logic rst,
                 input  logic mclk,
                 input  logic data);

  < Skriv SystemVerilog kode her >

endmodule
```

Oppgave 6 (vekt 6%)

I modulen *compute* som er vist under, utføres først en sammenligning b > c og deretter utføres beregningen av utgangsverdien *result* avhengig av resultatet av sammenligningen.

Det viser seg at VHDL koden ikke overholder timingkravet til FPGA-kretsen som skal brukes. Forsinkelsen gjennom modulen må nesten halveres for å overholde timingkravet. Dette kan løses ved at VHDLkoden endres til at det legges til en klokkeperiode med pipelining slik at utgangssignalet *result* kommer en klokkeperiode senere enn i arkitekturen vist under.

Lag en ny arkitektur *rtl2* til modulen *compute* med samme funksjon, men med en ekstra klokkeperiode pipelining som løser timingproblemet.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compute is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    a         : in  unsigned(3 downto 0);
    b         : in  unsigned(3 downto 0);
    c         : in  unsigned(3 downto 0);
    result    : out unsigned(3 downto 0));
end compute;

architecture rtl of compute is
begin

  process (rst, mclk) is
  begin
    if rst='1' then
      result <= (others => '0');
    elsif rising_edge(mclk) then
      if b > c then
        result <= a + b;
      else
        result <= a + c;
      end if;
    end if;
  end process;

end rtl;
```

Oppgave 7 (vekt 6%)

Modulen *rstmodule* som er vist under klokkes med klokkesignalet *mclk* og resettes med det asynchrone og aktivt høye resetsignalet *arst*.

Forklar **kort med ord** funksjonaliteten til prosessen P_RST og hvorfor componenten *bufg* inngår i modulen.

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.all;

entity rstmodule is
  port (
    arst        : in  std_logic;
    mclk        : in  std_logic;
    rst         : out std_logic);
end rstmodule;

architecture rtl of rstmodule is
  signal rst_s1, rst_s2 : std_logic;

  component bufg
    port(i : in std_logic;
         o : out std_logic);
  end component;

begin

  P_RST : process(arst, mclk) is
  begin
    if arst='1' then
      rst_s1 <= '1';
      rst_s2 <= '1';
    elsif rising_edge(mclk) then
      rst_s1 <= '0';
      rst_s2 <= rst_s1;
    end if;
  end process P_RST;

  bufg_inst: bufg
    port map (i => rst_s2,
              o => rst);

end rtl;
```

Oppgave 8 (vekt 6%)

I VHDL koden som er vist under, er det 5 feil/mangler.

Finn de 5 feilene/manglene og oppgi endringene i VHDL koden så feilene/manglene fjernes.

```
library ieee;
use ieee.std_logic_1164.all;

entity faulty is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    enable    : in  std_logic;
    data      : in  std_logic_vector(7 downto 0);
    equals    : out std_logic;
    term_cnt  : out std_logic);
end faulty;

architecture rtl of faulty is
  signal count: std_logic_vector(7 downto 0);
begin
  P_COMPARE: process(data) is
  begin
    if data=count then
      equals <= '1';
    else
      equals <= '0';
    end if;
  end process;

  P_COUNT: process(rst, mclk) is
  begin
    if rising_edge(mclk) then
      count <= count + 1;
    end if;
  end process;

  term_cnt <= 'Z' when enable='0' else
              '1' when count="1111" else
              '0';
end rtl;
```

Oppgave 9 (vekt 6%)

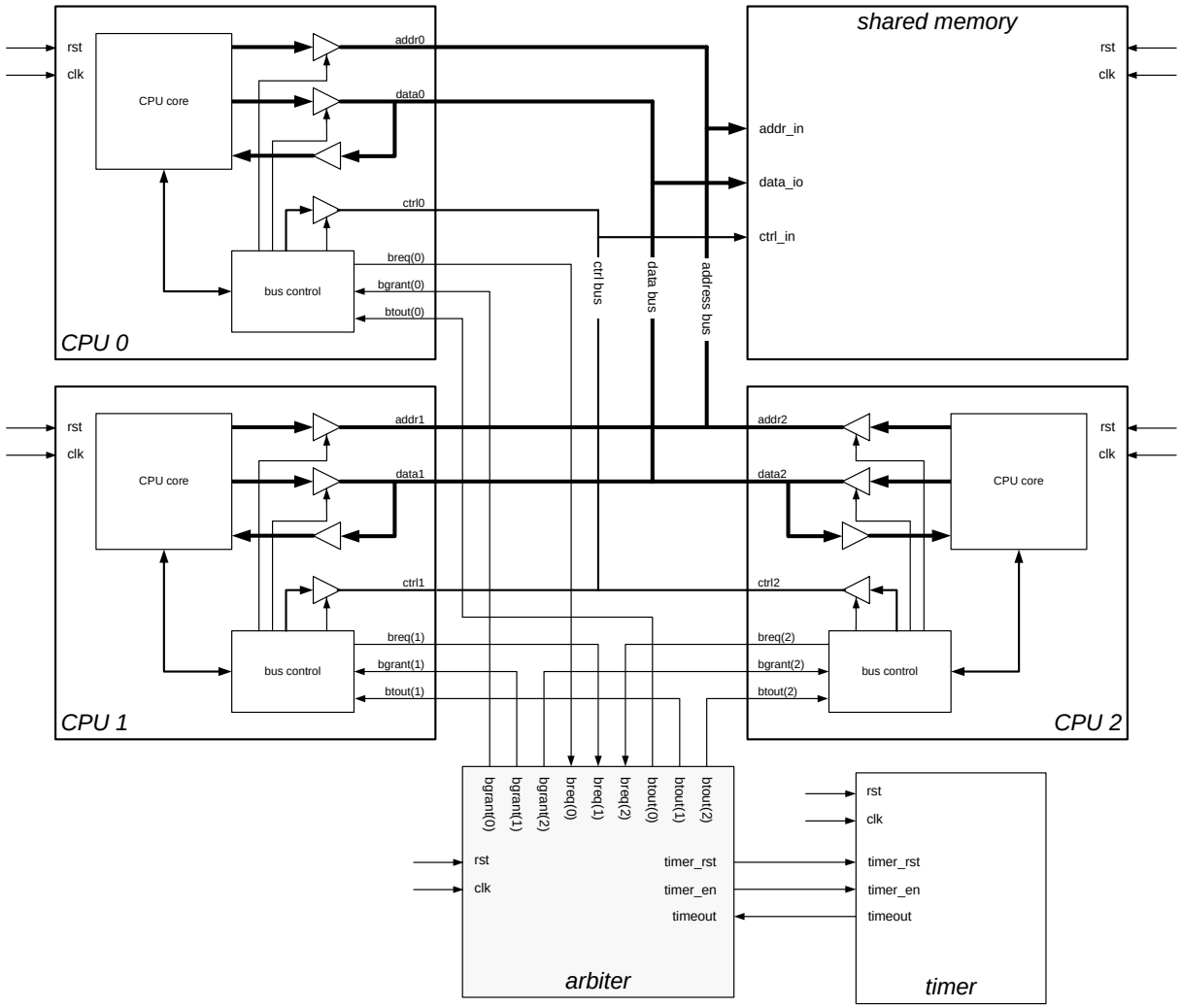
Under er det oppgitt entiteten og arkitekturen til modulen *something*.
Beskriv **kort med ord** funksjonaliteten til modulen *something*, og tegn et timingdiagram med alle signalene oppgitt i entiteten med påtrykksverdier til signalene *rst*, *mclk*, *send_str* og *din* de første 20 klokkeperiodene etter at reset blir inaktiv lav ('0') med signalet *din* med verdien x"13" slik at funksjonaliteten vises.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity something is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    send_str  : in  std_logic;
    din       : in  std_logic_vector(7 downto 0);
    dout      : out std_logic;
    sclk      : out std_logic;
    cs_n      : out std_logic);
end something;

architecture rtl of something is
  type state_type is (IDLE, START, PHASE0, PHASE1);
  signal present_state, next_state  : state_type;
  signal dreg, next_dreg      : std_logic_vector(7 downto 0);
  signal cnt, next_cnt        : unsigned(4 downto 0);
  signal next_dout            : std_logic;
  signal next_sclk            : std_logic;
  signal next_cs_n            : std_logic;
  signal dout_i               : std_logic;
  signal sclk_i               : std_logic;
  signal cs_i_n               : std_logic;
begin

  P_SEQ: process (rst, mclk) is
  begin
    if (rst = '1') then
      dout_i        <= '0';
      sclk_i        <= '1';
      cs_i_n        <= '1';
      dreg          <= (others => '0');
      cnt           <= (others => '0');
      present_state <= IDLE;
    elsif rising_edge(mclk) then
      dout_i        <= next_dout;
      sclk_i        <= next_sclk;
      cs_i_n        <= next_cs_n;
      dreg          <= next_dreg;
      cnt           <= next_cnt;
      present_state <= next_state;
    end if;
  end process P_SEQ;
```

```
P_COMB: process (present_state, send_str, din, cnt, dreg,
                 dout_i, sclk_i, cs_i_n) is
  begin
    next_dout   <= dout_i;
    next_sclk   <= sclk_i;
    next_cs_n   <= cs_i_n;
    next_dreg   <= dreg;
    next_cnt    <= cnt;
    next_state  <= present_state;

    case present_state is
      when IDLE =>
        next_sclk   <= '1';
        next_cs_n   <= '1';
        next_dout   <= '0';
        next_dreg   <= (others => '0');
        next_cnt    <= (others => '0');
        if send_str='1' then
          next_dreg   <= din;
          next_state  <= START;
        end if;
      when START =>
        next_cnt <= cnt + 1;
        if cnt(0)='1' then
          next_cs_n   <= '0';
          next_dout   <= dreg(7);
          next_dreg   <= dreg(6 downto 0) & '0';
          next_state  <= PHASE0;
        end if;
      when PHASE0 =>
        next_cnt <= cnt + 1;
        if cnt(0)='0' then
          next_sclk   <= '0';
          next_state  <= PHASE1;
        end if;
      when others =>
        next_cnt <= cnt + 1;
        if (cnt(4)='1' and cnt(0)='1') then
          next_sclk   <= '1';
          next_cs_n   <= '1';
          next_dout   <= '0';
          next_state <= IDLE;
        elsif cnt(0)='1' then
          next_sclk   <= '1';
          next_dout   <= dreg(7);
          next_dreg   <= dreg(6 downto 0) & '0';
          next_state  <= PHASE0;
        end if;
    end case;
  end process P_COMB;

  dout   <= dout_i;
  sclk   <= sclk_i;
  cs_n   <= cs_i_n;

end rtl;
```

Oppgave 10 (vekt 50%)

Vi skal i denne oppgaven se på en *bus arbiter*, norsk: *bus-arbitrerer***.**



Figur 1. Et multiprocessorsystem med 3 CPU*er og delt minne*, shared memory

I et multiprosessorsystem kan det være to eller flere mikroprosessorer, *CPU*er, som deler et felles minne, *shared memory*..
De aksesserer dette gjennom et bussystem som består av *address, data og ctrl bus (control bus).* De deler dermed alle disse signalene. Det er meget viktig at bare en enhet av gangen sender data ut på bussen. Derfor er bussignalene fra de enkelte *CPU*ene skilt fra bussen med tristate-buffere, se figuren over.

En *bus arbiter* er en enhet som til enhver tid har kontroll på hvilke enhet som skal ha tilgang til bussen. Dette styres gjennom en prosess som kalles "*bus arbitration*", norsk: *bus-arbitrering.*
Hver *CPU i, i=0,1,2* har to arbitreringssignaler: *breq(i) og bgrant(i)*[1]**.**
Når en *CPU* ønsker tilgang til bussen setter den *breq(i)*-signalet aktivt høyt ('1'). *bus-arbitrereren* svarer ved å sette *bgrant(i)*-signalet aktivt høyt ('1'). Det er ikke sikkert dette

[1] breq = bus request, bgrant = bus grant, btout = bus timeout

skjer med en gang fordi det kan være andre *CPU*er som ikke er ferdig med å benytte bussen og det kan være andre *CPU*er som også ønsker å benytte bussen, men har høyere prioritet. Når *bgrant(i)*-signalet går aktivt forteller det *CPU*en at den kan åpne sine tristate-buffere for å aksessere bussen. Når *CPU*en er ferdig med en overføring settes *breq(i)* -signalet inaktivt lavt ('0') og sørger med dette for at andre *CPU*er kan slippe til.

For å dele bussen rettferdig er det vanlig å endre prioriteten til de enkelt *breq(i)*-signalene dynamisk. Det betyr for eksempel at hvis vi har følgende synkende prioritet: *breq(0)> breq(1)> breq(2)* så vil *breq(0)* få aksess først uansett om *breq(1)* og/eller *breq(2)* er aktive samtidig. Etter at *breq(0)* settes inaktivt og aktivt igjen straks etterpå så har *breq(0)* fått laveste prioritet og kan ikke få tilgang til bussen før både *breq(1)* og *breq(2)* er inaktive.

Dette betyr at den som sist har hatt tilgang til bussen vil få lavest prioritet neste gang to eller flere *breq*-signaler er aktive samtidig. I vårt system kan vi kan ha følgende prioritetsendringer: *breq(0)>*[2]*breq(1)>breq(2)* endres til *breq(1)>breq(2)>breq(0)* endres til *breq(2)>breq(0)>breq(1)* som igjen endres til *breq(0)>breq(1)>breq(2)* osv…

I figuren under vises prioriteringen mellom *breq(0), breq(1)* og *breq(2)* når alle *breq*-signalene settes aktivt høye ('1') samtidig. Deretter settes *breq(i)* aktivt høyt igjen med en gang *bgrant(i)* har blitt inaktivt. Dermed vises prioriteringen mellom *breq(i)*-signalene.

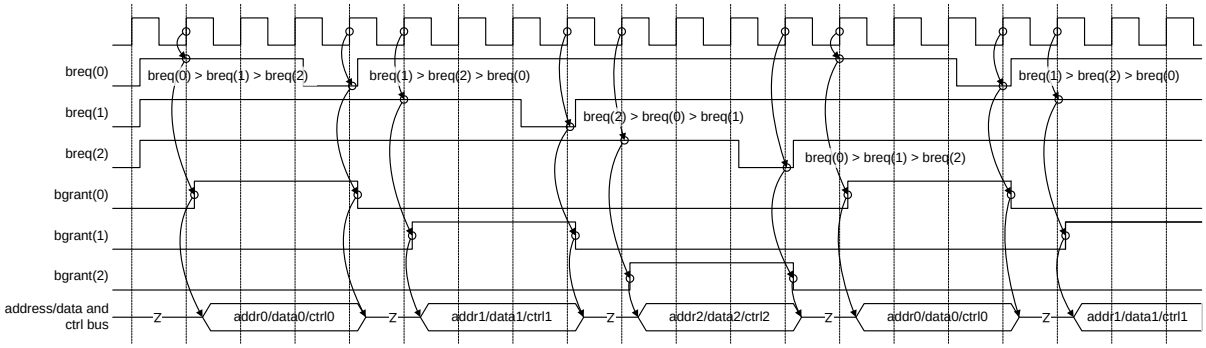


Figure 2. Bus-request og prioritering

I tillegg skal man sørge for at hver *CPU* ikke kan bruke for lang tid på bussen. Dette gjøres ved å aktivere en timer med *timer_en* (aktivt høyt) i tilstanden som *bgrant(i)* er aktivt. Dersom *timeout* går aktivt (høyt) så har man en timeoutsituasjon som flagges med at signalet *btout(i)* går aktivt høyt *og* lagres i et register, samtidig med at *bgrant(i)* settes inaktivt lavt. Når *btout(i)* går aktivt skal *CPU i* svare ved å sette *breq(i)* inaktivt lavt i neste klokkeperiode; se figur 3. *btout(i)* resettes neste gang *breq(i)* har gått aktivt og *CPU i* har fått tilgang til bussen (norsk-engelsk:"*grantet*" bussen). *timer*-modulen er vist i figur 1.

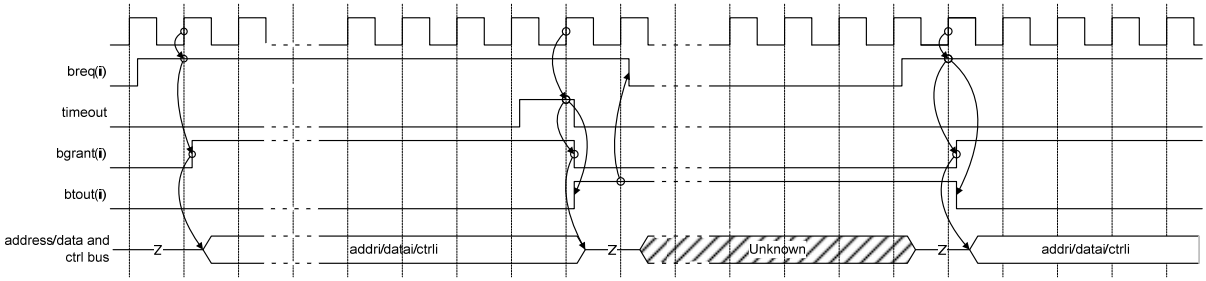


Figure3. Bus-request med timeout

[2] ">" betyr i denne sammenhengen større prioritet enn

a) Vekt 15%
Lag et ASM-flytdiagrammet som beskriver en tilstandsmaskin, *arbiter*, som arbitrerer mellom de tre *CPU*ene i figuren over og med *timeout* -funksjonen. Tilstandsmaskinen skal beskrives som en Moore-maskin**.**

Benytt følgende entitet:

```
entity arbiter is
  port
    (
      rst       : in  std_logic;
      clk       : in  std_logic;
      breq      : in  std_logic_vector(2 downto 0);
      bgrant    : out std_logic_vector(2 downto 0);
      btout     : out std_logic_vector(2 downto 0);
      timer_rst : out std_logic;
      timer_en  : out std_logic;
      timeout   : in  std_logic
      );
end entity arbiter;
```

b) Vekt 15%
Implementer tilstandsmaskinen i a) som en to-process VHDL-beskrivelse.

c) Vekt 15%
Lag en testbenk som tester tilstandsmaskinen beskrevet i b).

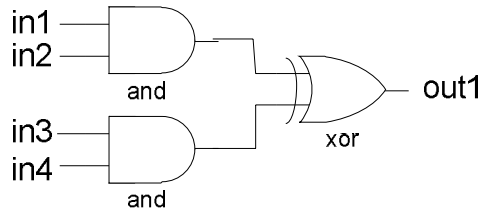
d) Vekt 5%
Gjøre rede for prinsippene for en selvsjekkende (selvtestende) testbenk. Beskriv kort med ord hva som skal til for å gjøre testbenken i punkt c) selvsjekkende.

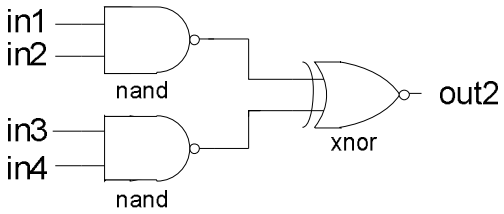
INF4431. Oppgavesvar for kandidat nr: _____

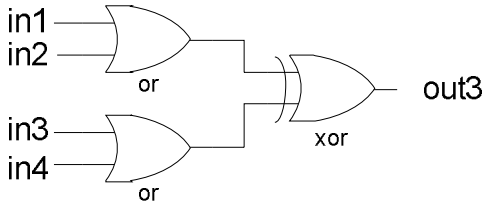
Oppgave	A	B	C	D	E
1					
2					
3					
4					

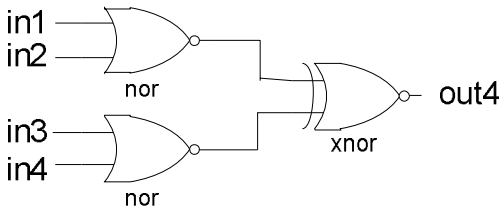
Oppgave 1

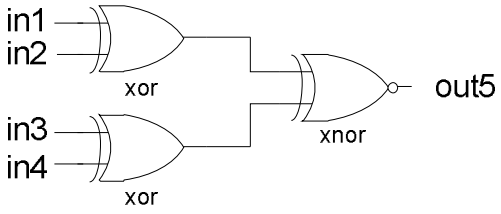
Figuren under viser de kombinatoriske kretsene med out1 lik “and-and-xor”, out2 lik “nand-nand-xnor”, out3 lik “or-or-xor”, out4 lik “nor-nor-xnor” og out5 lik “xor-xor-xnor”.











En 4-input Xilinx Spartan-3 LUT med innhold **”7888”** (hex) realiserer en:	A	and-and-xor	X
	B	nand-nand-xnor	
	C	or-or-xor	
	D	nor-nor-xnor	
	E	xor-xor-xnor	

Oppgave 2

Kan variabler i VHDL deklareres i:	A	Prosess	X
	B	Architecture	
	C	Procedure	X
	D	Function	X
	E	Entity	

Oppgave 3

Konstruksjon 1	A	Xilinx har harde DSP moduler som har MAC (Multiply and Accumulate) funksjon.	X
	B	Block RAM (BRAM) i FPGA har ikke en kjent initialverdi etter konfigurering.	
	C	Bruk av dedikert mentelogikk i FPGA'en gjør ikke at det blir mindre tilgjengelig logikk i FPGA'en.	X
	D	Ved synkron styring av flip-flop'er med set/reset er det best at set kommer før reset i VHDL koden.	
	E	I Spartan-3 teknologi har forbindelseslinjer mellom LUT'er vanligvis mindre tidsforsinkelse enn gjennom LUT'er.	

Oppgave 4

Konstruksjon 2	A	Det er raskere å simulere med variabler enn med signaler i VHDL.	X
	B	For hver delta cycle i en VHDL simulator går tiden 1 nanosekund.	
	C	Selv om antall input til en funksjon er konstant øker forbruket av LUT'er med kompleksiteten til funksjonen.	
	D	En Xilinx Block RAM (BRAM) har to porter som kan leses fra og skrives til i samme klokkeperiode.	X
	E	Ved delvis rekonfigurasjon vil de delene av kretsen som ikke blir rekonfigurert også være aktive under rekonfigurering.	X

Oppgave 5 (for INF3430)

Det er ikke krav om testbenk i besvarelsen. Testbenken er kun tatt med for enklere kunne simulere besvarelsen.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity strobegen is
  port (
    rst         : in  std_logic;
    mclk        : in  std_logic;
    data        : in  std_logic;
    rising_str  : out std_logic;
    falling_str : out std_logic;
    cycle_cnt   : out std_logic_vector(15 downto 0));
end strobegen;

architecture rtl of strobegen is

  signal data_s1, data_s2, data_d1 : std_logic;

begin

  process (rst, mclk)
    variable cycle_cnt_ena : std_logic;
    variable cycle_cnt_i   : unsigned(15 downto 0);
  begin
    if rst='1' then
      data_s1       <= '0';
      data_s2       <= '0';
      data_d1       <= '0';
      rising_str    <= '0';
      falling_str   <= '0';
      cycle_cnt     <= (others => '0');
      cycle_cnt_ena := '0';
      cycle_cnt_i   := (others => '0');

    elsif rising_edge(mclk) then

      data_s1 <= data;
      data_s2 <= data_s1;
      data_d1 <= data_s2;

      rising_str <= '0';
      if data_s2='1' and data_d1='0' then
        rising_str    <= '1';
        cycle_cnt_ena := '1';
        cycle_cnt_i   := (others => '0');
      end if;

      falling_str <= '0';
```

```
      if data_s2='0' and data_d1='1' then
        falling_str   <= '1';
        cycle_cnt_ena := '0';
        cycle_cnt     <= std_logic_vector(cycle_cnt_i);
      end if;

      if cycle_cnt_ena='1' and cycle_cnt_i < x"FFFF" then
        cycle_cnt_i := cycle_cnt_i + 1;
      end if;

    end if;
  end process;
end rtl;


library ieee;
use ieee.std_logic_1164.all;

entity tb_strobegen is
  -- empty;
end tb_strobegen;

architecture beh of tb_strobegen is

  component strobegen is
    port (rst         : in  std_logic;
          mclk        : in  std_logic;
          data        : in  std_logic;
          rising_str  : out std_logic;
          falling_str : out std_logic;
          cycle_cnt   : out std_logic_vector(15 downto 0));
  end component strobegen;

  signal rst         : std_logic;                     -- [in]
  signal mclk        : std_logic;                     -- [in]
  signal data        : std_logic;                     -- [in]
  signal rising_str  : std_logic;                     -- [out]
  signal falling_str : std_logic;                     -- [out]
  signal cycle_cnt   : std_logic_vector(15 downto 0);  -- [out]

begin

  strobegen_inst: strobegen
    port map (rst         => rst,          -- [in]
              mclk        => mclk,         -- [in]
              data        => data,         -- [in]
              rising_str  => rising_str,   -- [out]
              falling_str => falling_str,  -- [out]
              cycle_cnt   => cycle_cnt);   -- [out]

  P_CLOCK: process is
  begin
      mclk <= '0';
```

```
      wait for 50 ns;
      mclk <= '1';
      wait for 50 ns;
  end process P_CLOCK;

  rst <= '1', '0' after 100 ns;

--  data <= '0',
--          '1' after 200 ns,
--          '0' after 600 ns,
--          '1' after 10000 ns,
--          '0' after 1100 ns;

  data <= '0',
          '1' after 200 ns,
          '0' after 600 ns,
          '1' after 700 ns,
          '0' after 6700000 ns,
          '1' after 6700100 ns,
          '0' after 6700200 ns;

end beh;
```

Oppgave 5 (for INF4431)

SystemVerilog modulen strobegen kan simuleres med samme VHDL testbenk som oppgitt for VHDL løsningen.

```
module strobegen(output logic rising_str,
                 output logic falling_str,
                 output logic [15:0] cycle_cnt,
                 input  logic rst,
                 input  logic mclk,
                 input  logic data);

   logic data_s1, data_s2, data_d1;

   logic cycle_cnt_ena;
   logic [15:0] cycle_cnt_i;

   always_ff @(posedge mclk, negedge rst)
   begin
     if (rst) begin
       data_s1 <= '0;
       data_s2 <= '0;
       data_d1 <= '0;
     end else begin
       data_s1 <= data;
       data_s2 <= data_s1;
       data_d1 <= data_s2;
     end;
   end // always_ff @ (posedge mclk, negedge rst)

   always_ff @(posedge mclk, negedge rst)
   begin
     if (rst) begin
       rising_str    <= '0;
       falling_str   <= '0;
       cycle_cnt     <= '0;
       cycle_cnt_ena  = '0;
       cycle_cnt_i    = '0;
     end else begin

       if (data_s2 && ~data_d1) begin
         rising_str    <= '1;
         cycle_cnt_ena  = '1;
         cycle_cnt_i    = '0;
       end else
         rising_str <= '0;

       if (~data_s2 && data_d1) begin
         falling_str   <= '1;
         cycle_cnt_ena  = '0;
         cycle_cnt     <= cycle_cnt_i;
       end else
         falling_str <= '0;
```

```
        if (cycle_cnt_ena && cycle_cnt_i < 16'hFFFF)
          cycle_cnt_i = cycle_cnt_i + 1;

      end; // else: !if(rst)

   end // always_ff @ (posedge mclk, negedge rst)

endmodule
```

Oppgave 6

Det er ikke krav om testbenk i besvarelsen. Testbenken er kun tatt med for enklere kunne simulere besvarelsen.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity compute is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    a         : in  unsigned(7 downto 0);
    b         : in  unsigned(7 downto 0);
    c         : in  unsigned(7 downto 0);
    result    : out unsigned(7 downto 0));
end compute;

architecture rtl of compute is

  signal b_gt_c : std_logic;

  -- For answer 2:
  signal a_d1, b_d1, c_d1 : unsigned(7 downto 0);

  -- For answer 3:
  signal result_pp1, result_pp2 : unsigned(7 downto 0);

begin

-- Answer 1:

  process (rst, mclk)
  begin
    if rst='1' then
      b_gt_c <= '0';
      a_d1   <= (others => '0');
      b_d1   <= (others => '0');
      c_d1   <= (others => '0');
   elsif rising_edge(mclk) then
      b_gt_c <= '0';
      if b > c then
        b_gt_c <= '1';
      end if;
      a_d1 <= a;
      b_d1 <= b;
      c_d1 <= c;
    end if;
  end process;

  process (rst, mclk)
  begin
```

```
    if rst='1' then
      result <= (others => '0');
    elsif rising_edge(mclk) then
      if b_gt_c = '1' then
        result <= a_d1 + b_d1;
      else
        result <= a_d1 + c_d1;
      end if;
    end if;
  end process;

-- Answer 2:

--  process (rst, mclk)
--  begin
--    if rst='1' then
--      b_gt_c <= '0';
--      a_d1   <= (others => '0');
--      b_d1   <= (others => '0');
--      c_d1   <= (others => '0');
--      result <= (others => '0');
--    elsif rising_edge(mclk) then

--      b_gt_c <= '0';
--      if b > c then
--        b_gt_c <= '1';
--      end if;
--      a_d1 <= a;
--      b_d1 <= b;
--      c_d1 <= c;

--      if b_gt_c = '1' then
--        result <= a_d1 + b_d1;
--      else
--        result <= a_d1 + c_d1;
--      end if;

--    end if;

--  end process;

-- Answer 3:

--  process (rst, mclk)
--  begin
--    if rst='1' then
--      b_gt_c     <= '0';
--      result_pp1 <= (others => '0');
--      result_pp2 <= (others => '0');
--      result     <= (others => '0');
--    elsif rising_edge(mclk) then

--      b_gt_c <= '0';
```

```
--      if b > c then
--        b_gt_c <= '1';
--      end if;
--      result_pp1 <= a + b;
--      result_pp2 <= a + c;

--      if b_gt_c = '1' then
--        result <= result_pp1;
--      else
--        result <= result_pp2;
--      end if;

--    end if;

--  end process;

end rtl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_compute is
  -- empty;
end tb_compute;

architecture beh of tb_compute is

  component compute is
    port (rst    : in  std_logic;
          mclk   : in  std_logic;
          a      : in  unsigned(7 downto 0);
          b      : in  unsigned(7 downto 0);
          c      : in  unsigned(7 downto 0);
          result : out unsigned(7 downto 0));
  end component compute;

  signal rst    : std_logic;             -- [in]
  signal mclk   : std_logic;             -- [in]
  signal a      : unsigned(7 downto 0);  -- [in]
  signal b      : unsigned(7 downto 0);  -- [in]
  signal c      : unsigned(7 downto 0);  -- [in]
  signal result : unsigned(7 downto 0);  -- [out]

begin

  compute_inst: compute
    port map (rst    => rst,            -- [in]
              mclk   => mclk,           -- [in]
              a      => a,              -- [in]
              b      => b,              -- [in]
              c      => c,              -- [in]
              result => result);        -- [out]
```

```
  P_CLOCK: process is
  begin
      mclk <= '0';
      wait for 50 ns;
      mclk <= '1';
      wait for 50 ns;
  end process P_CLOCK;

  rst  <= '1', '0' after 100 ns;

  a <= x"00",
       x"01" after 200 ns,
       x"02" after 400 ns;

  b <= x"00",
       x"02" after 200 ns,
       x"04" after 400 ns;

  c <= x"00",
       x"01" after 200 ns,
       x"06" after 400 ns;

end beh;
```

Oppgave 7

Prosessen P_RST vil asynkront uavhengig av klokken mclk aktivere signalet rst (dvs. sette rst til ‘1’) og synkront med klokken mclk deaktivere rst (dvs. sette signalet til ‘0’).

Componenten bufg inngår i modulen for at rst signalet kan distribueres med liten tidsforsinkelse til veldig mange reset innganger på flip-flop’er i andre moduler.

Oppgave 8

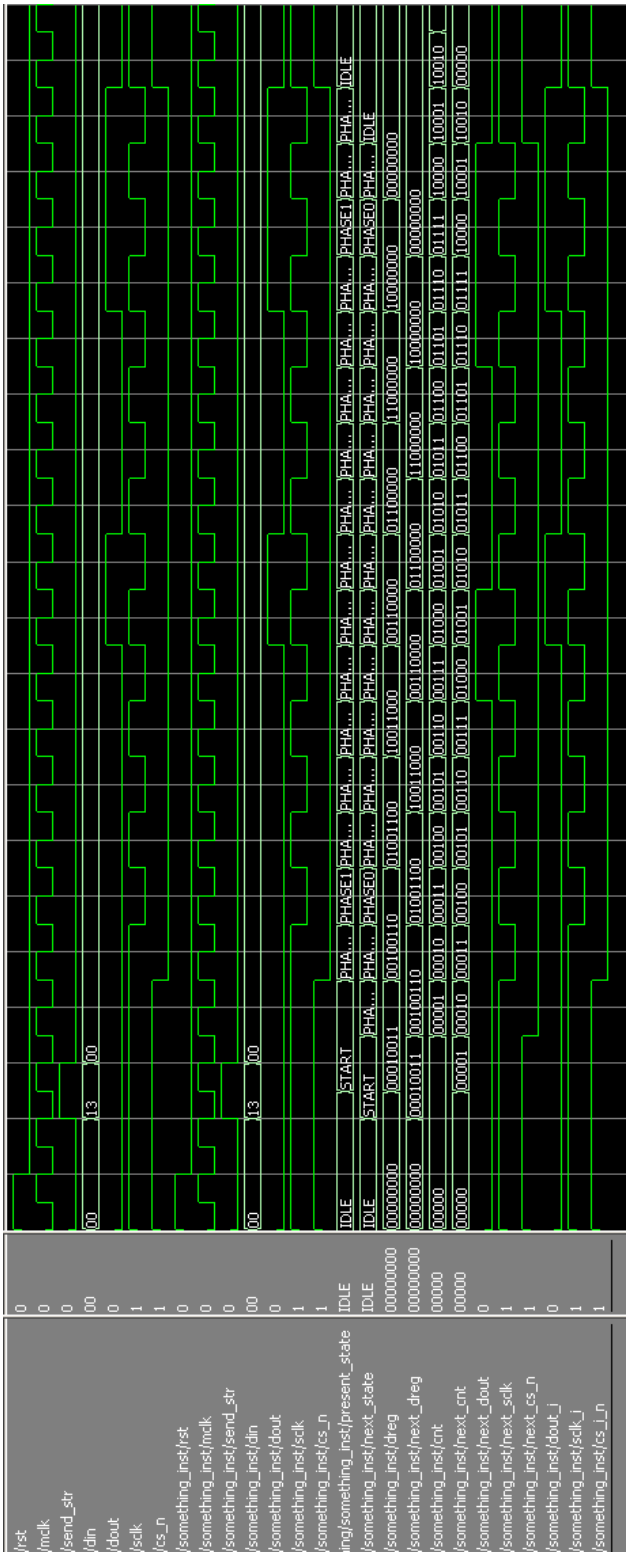
I VHDL koden som er vist under er de 5 feil/mangler vist understreket med bold type.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity faulty is
  port (
    rst       : in  std_logic;
    mclk      : in  std_logic;
    enable    : in  std_logic;
    data      : in  std_logic_vector(7 downto 0);
    equals    : out std_logic;
    term_cnt  : out std_logic);
end faulty;

architecture rtl_answ of faulty is
  signal count: std_logic_vector(7 downto 0);
begin
  P_COMPARE: process(count, data)
  begin
    if data=count then
      equals <= '1';
    else
      equals <= '0';
    end if;
  end process;

  P_COUNT: process(rst, mclk)
  begin
    if rst='1' then
      count <= "11111111";
    elsif rising_edge(mclk) then
      count <= std_logic_vector(unsigned(count) + 1);
    end if;
  end process;

  term_cnt <= 'Z' when enable='0' else
              '1' when count="11111111" else
              '0';
end rtl;
```

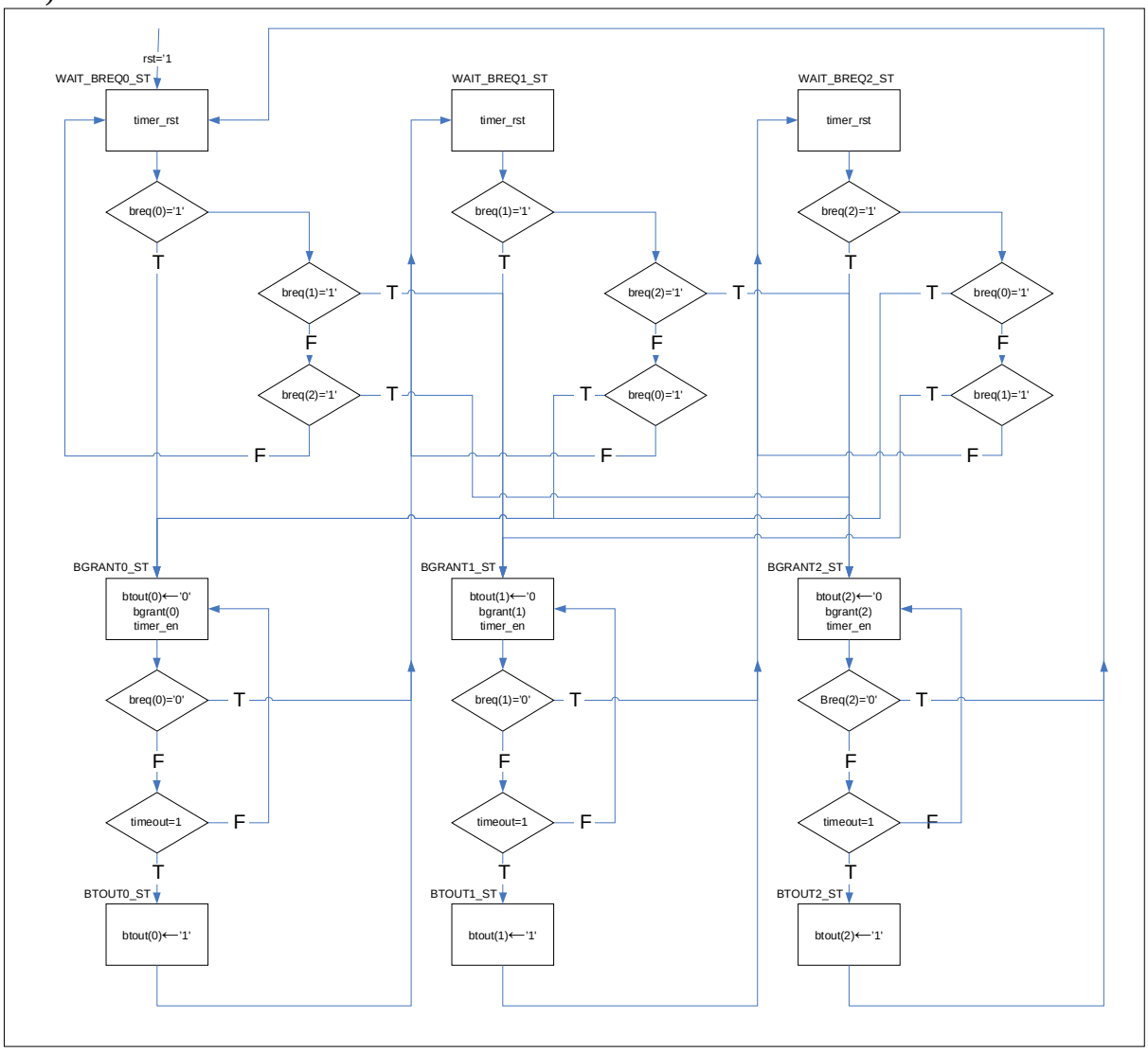
Oppgave 9

Modulen something er en såkalt SPI (Serial Port Interface) modul som skifter inn 8 bit når send_str er aktiv høy og skifter disse bitene ut med MSB bitet først i takt med et generert klokkesignal kalt sclk når sclk går fra ‘0’ til ‘1’. Signalet cs_n viser når data blir skiftet ut.

Detaljert timing er vist i timingdiagrammet under:



10a)



10b)

```
library IEEE;
use IEEE.std_logic_1164.all;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity arbiter is
  port
    (
      rst       : in  std_logic;
      clk       : in  std_logic;
      breq      : in  std_logic_vector(2 downto 0);
      bgrant    : out std_logic_vector(2 downto 0);
      btout     : out std_logic_vector(2 downto 0);
      timer_rst : out std_logic;
      timer_en  : out std_logic;
```

```
      timeout   : in  std_logic
      );
end entity arbiter;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

architecture rtl_arbiter of arbiter is

  type arbiter_type_st is (WAIT_BREQ0_ST,WAIT_BREQ1_ST,
                           WAIT_BREQ2_ST,BGRANT0_ST,BGRANT1_ST,
                           BGRANT2_ST,BTOUT0_ST,BTOUT1_ST,BTOUT2_ST);
  signal curr_st, next_st : arbiter_type_st;
  signal curr_btout, next_btout  : std_logic_vector(2 downto 0);

begin

  STATE_REG:
  process(rst, clk)
  begin
    if rst = '1' then
      curr_btout <= (others => '0');
      curr_st   <= WAIT_BREQ0_ST;
    elsif rising_edge(clk) then
      curr_btout <= next_btout;
      curr_st   <= next_st;
    end if;
  end process STATE_REG;

  btout <= curr_btout;

  STATE_COMB:
  process(breq, timeout, curr_st, curr_btout)
  begin
    timer_en  <= '0';
    timer_rst <= '0';
    bgrant    <= (others => '0');
    next_btout <= curr_btout;
    next_st   <= curr_st;

    case curr_st is

      when WAIT_BREQ0_ST =>
        timer_rst <= '1';
        if breq(0) = '1' then
          next_st <= BGRANT0_ST;
        elsif breq(1) = '1' then
          next_st <= BGRANT1_ST;
        elsif breq(2) = '1' then
          next_st <= BGRANT2_ST;
        end if;
      when BGRANT0_ST =>
```

```
  timer_en  <= '1';
  bgrant(0) <= '1';
  next_btout(0) <= '0';
  if breq(0) = '0' then
    next_st <= WAIT_BREQ1_ST;
  elsif timeout = '1' then
    next_st <= BTOUT0_ST;
  end if;
when BTOUT0_ST =>
  next_btout(0) <= '1';
  next_st <= WAIT_BREQ1_ST;

when WAIT_BREQ1_ST =>
  timer_rst <= '1';
  if breq(1) = '1' then
    next_st <= BGRANT1_ST;
  elsif breq(2) = '1' then
    next_st <= BGRANT2_ST;
  elsif BREQ(0) = '1' then
    next_st <= BGRANT0_ST;
  end if;
when BGRANT1_ST =>
  timer_en  <= '1';
  bgrant(1) <= '1';
  next_btout(1) <= '0';
  if breq(1) = '0' then
    next_st <= WAIT_BREQ2_ST;
  elsif timeout = '1' then
    next_st <= BTOUT1_ST;
  end if;
when BTOUT1_ST =>
  next_btout(1) <= '1';
  next_st <= WAIT_BREQ2_ST;

when WAIT_BREQ2_ST =>
  timer_rst <= '1';
  if breq(2) = '1' then
    next_st <= BGRANT2_ST;
  elsif breq(0) = '1' then
    next_st <= BGRANT0_ST;
  elsif breq(1) = '1' then
    next_st <= BGRANT1_ST;
  end if;
when bgrant2_st =>
  timer_en  <= '1';
  bgrant(2) <= '1';
  next_btout(2) <= '0';
  if breq(2) = '0' then
    next_st <= WAIT_BREQ0_ST;
  elsif timeout = '1' then
    next_st <= BTOUT2_ST;
  end if;
when btout2_st =>
```

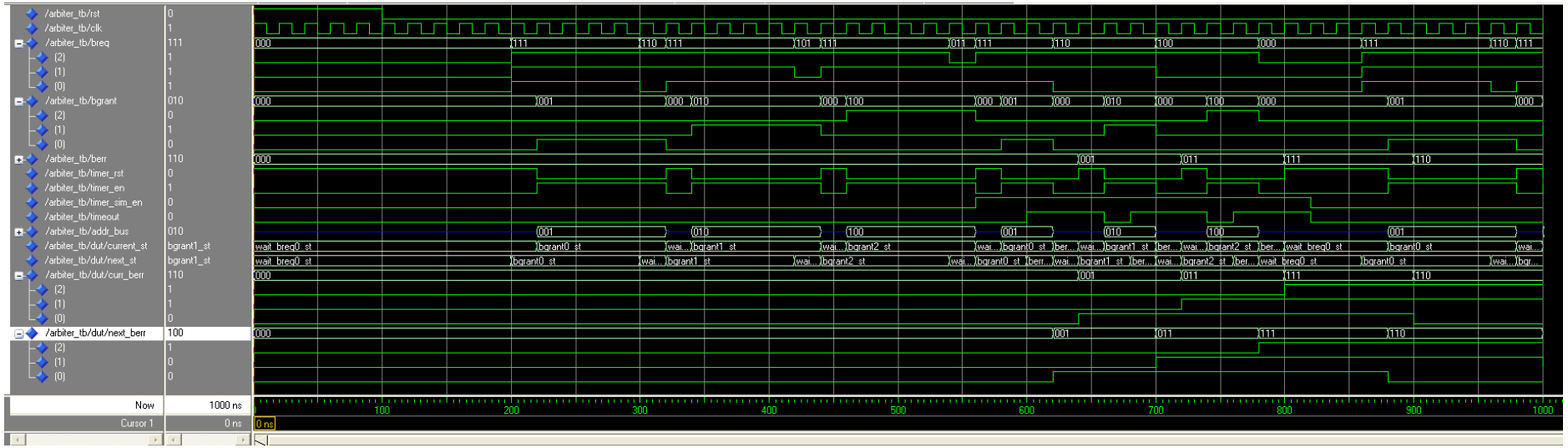
```
        next_btout(2) <= '1';
        next_st <= WAIT_BREQ0_ST;
    end case;
  end process STATE_COMB;

end architecture RTL_ARBITER;
```

10c)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity arbiter_tb is
end arbiter_tb;

architecture test_arbiter of arbiter_tb is

  component arbiter
    port (
      rst       : in  std_logic;
      clk       : in  std_logic;
      breq      : in  std_logic_vector(2 downto 0);
      bgrant    : out std_logic_vector(2 downto 0);
      btout     : out std_logic_vector(2 downto 0);
      timer_rst : out std_logic;
      timer_en  : out std_logic;
      timeout   : in  std_logic);
  end component;

  -- component ports
  signal rst          : std_logic;
  signal clk          : std_logic := '1';
  signal breq         : std_logic_vector(2 downto 0) := (others => '0');
  signal bgrant       : std_logic_vector(2 downto 0);
  signal btout        : std_logic_vector(2 downto 0) := (others => '0');
  signal timer_rst    : std_logic;
  signal timer_en     : std_logic;
  signal timeout      : std_logic := '0';

  constant TCLK       : time := 20 ns;

  signal timer_sim_en : std_logic := '0';
  signal addr_bus  : std_logic_vector(2 downto 0); --to visualize the
tristate function

begin  -- TEST_ARBITER

  -- component instantiation
  dut: arbiter
    port map (
      rst       => rst,
      clk       => clk,
```

```
    breq      => breq,
    bgrant    => bgrant,
    btout     => btout,
    timer_rst => timer_rst,
    timer_en  => timer_en,
    timeout   => timeout);

-- clock generation
clk <= not clk after TCLK/2;
addr_bus <= bgrant when (unsigned(bgrant) > 0) else (others => 'Z');

-- waveform generation
wavegen_proc: process
begin
  -- insert signal assignments here
  rst <= '1','0' after 100 ns;
  timer_sim_en <= '0';
  wait for tclk*10;

  report "testing the priorities";
  breq <= "111";
  for i in 0 to 2 loop
    wait until bgrant'event;
    wait for tclk*4;
    breq(i) <= '0';
    wait until bgrant'event;
    breq(i) <= '1';
  end loop;

  report "testing the timeout mechanism";
  breq <= "111";
  timer_sim_en <= '1';
  for i in 0 to 2 loop
    wait until bgrant'event;
    wait until timeout'event;
    wait for tclk;
    breq(i) <= '0';
    wait for tclk*2;
  end loop;

  report "testing the btout signals going off";
  timer_sim_en <= '0';
  wait for tclk*2;
  breq <= "111";
  for i in 0 to 2 loop
    wait until bgrant'event;
    wait for tclk*4;
    breq(i) <= '0';
    wait until bgrant'event;
    breq(i) <= '1';
  end loop;
  wait;
end process wavegen_proc;

TIMEOUT_SIM:
process(rst,clk)
```

```
  begin
    if rst = '1' then
      timeout <= '0';
    elsif rising_edge(clk) then
      if timer_rst = '1' then
        timeout <= '0';
      elsif timer_en = '1' and timer_sim_en = '1' then
        timeout <= '1';
      end if;
    end if;
  end process;
end test_arbiter;


configuration arbiter_tb_test_arbiter_cfg of arbiter_tb is
  for test_arbiter
  end for;
end arbiter_tb_test_arbiter_cfg;
```


/arbiter_tb/rst
/arbiter_tb/clk
/arbiter_tb/breq
/arbiter_tb/bgrant
/arbiter_tb/berr
/arbiter_tb/timer_rst
/arbiter_tb/timer_en
/arbiter_tb/timer_sim_en
/arbiter_tb/timeout
/arbiter_tb/addr_bus
/arbiter_tb/dut/current_st
/arbiter_tb/dut/next_st
/arbiter_tb/dut/curr_berr
/arbiter_tb/dut/next_berr
Now
1000 ns
Cursor 1
0 ns

10d)

Prinsippende bak en selvtestende(eller selvsjekkende testbenk) er at utgangssignalene til UUT sammenlignes med forventede verdier (fasit) i fra testbenken. Det betyr at vi ikke trenger å granske Waveforms for å finne ut om UUT fungerer riktig eller ikke.
Fasiten kan leses fra en fil eller den kan være en del av selve testbenken.
Det er vanlig å lage en eller flere (overloadede) ***check_val(<signals to be checked>, <expected result>)*** prosedyrer som har som input de aktuelle signalene som skal sjekkes og forventede verdier (fasit) på disse signalene. Det er vanlig prosedyren vedlikeholder en feilteller og skriver ut en rapport, gjerne med et tidsstempel, når feil blir funnet. Etter simuleringen er ferdig skrives det ut en sluttrapport som viser om simulingen er OK eller ikke OK.

I testbenken i oppgave c) er det aktuelt å skrive en check_val-prosedyre med med utgangssignalene fra arbiter og fasit for disse som input. Man kan kalle disse etter først å ha gitt stimuli på bregq-inngangene og så vente på en bgrant eller en btout event og sammenligne resultane av bgrant og btout med forventet verdi på disse.

```
--Pseudokode av stimuliprosessen:
STIMULI:
process
  procedure check_val(barbiter_out : in std_logic_vector;
                      fasit        : in std_logic_vector) is
  begin
    if barbiter_out /= fasit then
      error := error + 1;
      report "Error found at time: " & time'image(now);
    end if;
  end procedure;
  signal barbiter_out : std_logic_vector(7 downto 0),
begin
  .
  .
  barbiter_out <= bgrant & breq & timeout_rst & timeout_en;
  for i in 0 to cnt loop
    breq <= <new_breq(i)>
    wait until barbiter_out'event;
    check_val(barbiter_out,<barbiter_out_fasit(2*i)>);
    wait until barbiter_out'event;
    check_val(barbiter_out,<barbiter_out_fasit(2*i+1)>);
  end loop;

  if error = 0 then
    report("Simulation OK");
  else
    report("Simulation not OK");
  end if;
  wait;
end process;
```

INF3430/4431. Fasit oppgave 1-4

Oppgave	A	B	C	D	E
1	X				
2	X		X	X	
3	X		X		
4	X			X	X