UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	**INF3430/4431**
Eksamensdag:	**1. Desember 2017**
Tid for eksamen:	**09.00-13.00**
Oppgavesettet er på 11 sider	
Vedlegg:	**1**
Tillatte hjelpemidler:	**Alle trykte og skrevne, samt kalkulator**

Kontroller at oppgavesettet er komplett før du begynner å besvare spørsmålene

Oppgaveteksten består av oppgave 1–5, 9-10 (flervalgsoppgaver) som skal besvares på skjemaet som er vedlagt etter oppgaveteksten og oppgave 6-8 og 11 som besvares på vanlige ark.

Oppgavenes vekt er vist i parentes bak oppgavenes nummer.

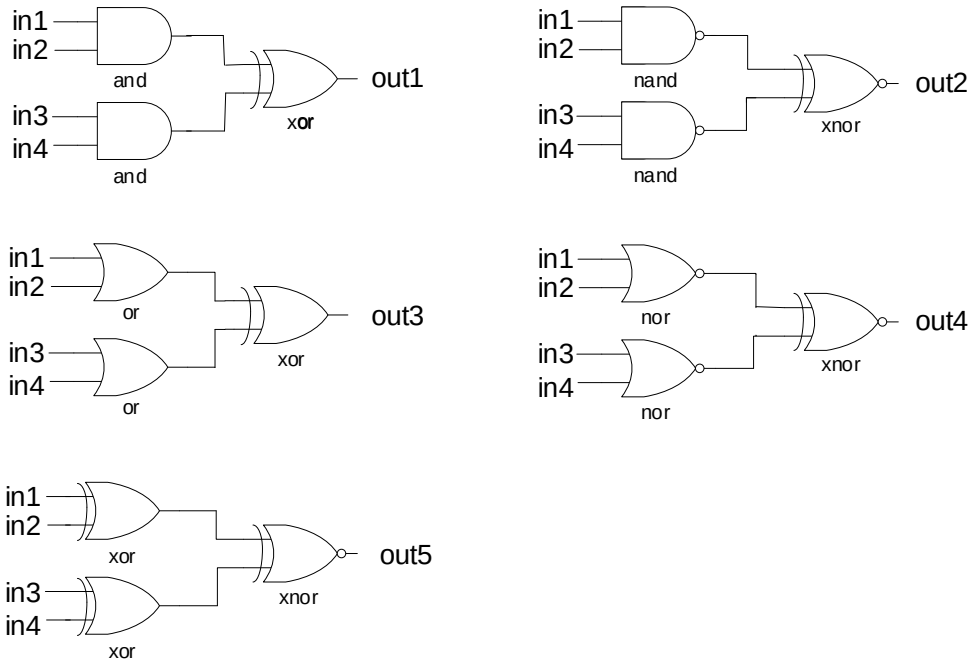
Oppgavene er uavhengige så de kan løses hver for seg.

I VHDL-oppgavene er det i besvarelsen ikke nødvendig å gjenta VHDL kode som allerede står i oppgaveteksten.

Generelt for oppgave 1–5, 9-10:
Hver oppgave består av et tema 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 vedlegg 1. 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 vedlegg 1 er din besvarelse.

Oppgave 1 (3 %)

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 LUT med INIT verdi lik **”111E”** (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 (3 %)

FPGA teknologi	A	Når en FPGA er konfigurert er tilstanden i Block RAM (BRAM) ukjent.	
	B	Xilinx Block RAM’er kan ikke fjernes fra FPGA’en for å spare plass.	
	C	Xilinx BRAM kan brukes som ROM.	
	D	RAM kan lages av Xilinx LUT’er.	
	E	I VHDL-2008 kan sensitivity listen til en kombinatorisk prosess skrives som «all», dvs. process(all)	

Oppgave 3 (3 %)

FPGA teknologi	A	MicroBlaze er en hard prosessorkjerne	
	B	I Zynq-7000 er det en ARM hard prosessorkjerne	
	C	En myk prosessorkjerne krever mindre plass enn en hard prosessorkjerne	
	D	BFM (Bus Functional Model) kan bare brukes i testbenker	
	E	Gigabit Transceivere finnes som myke kjerner.	

Oppgave 4 (3 %)

FPGA teknologi	A	Det er enkelt å oppdage metastabilitet ved simulering.	
	B	Etter en tid i metastabil tilstand vil alle flip-flop'er alltid gå tilbake til '0'.	
	C	En BUFG modul kan bare brukes til klokkesignaler.	
	D	Initialverdien etter deklarasjon av et signal av typen std_logic vil være 'U'.	
	E	To signaler av typen std_logic med verdiene '0' og '1' som driver samme signal får verdien 'Z'.	

Oppgave 5 (3 %)

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

Oppgave 6 (8%)

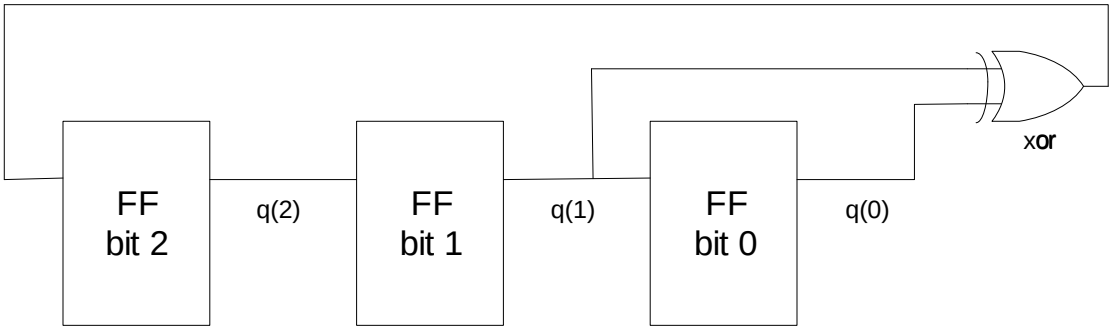
Det skal i denne oppgaven lages en MOORE type FSM hvor utgangen q genererer tallene 4, 2, 5, 6, 7, 3, 1 og deretter genererer samme rekke av tall igjen så lenge signalet run er aktivt høyt (‘1’). Når signalet run er lavt (‘0’) skal utgangen q beholde det sist genererte tallet til signalet run blir aktivt høyt (‘1’) igjen og genereringen fortsetter. Når reset signalet rst er aktivt høyt skal utgangen q ha verdien 4 som er første verdi i sekvensen.

Implementer arkitekturen til modulen arbitrary_sequence_gen som er vist under i syntetiserbar VHDL.

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

entity arbitrary_sequence_gen is
  port
    (
      rst   : in  std_logic;
      clk   : in  std_logic;
      run   : in  std_logic;
      q     : out std_logic_vector(2 downto 0)
    );
end entity arbitrary_sequence_gen;

architecture rtl of arbitrary_sequence_gen is

  < Skriv deklarasjoner her >

begin

  < Lag stateregister prosess her >

  < Lag neste tilstand prosess her >

end architecture rtl;
```

Oppgave 7 (10%)

Det skal i denne oppgaven lages en Linear Feedback Shift Register (LFSR) modul med synkron klokket XOR tilbakekobling til et 3 bit register som vist i figuren under.



Legg merke til at det det utføres en xor mellom bit 0 og bit 1 og at resultatet av xor føres inn på inngangen til bit 2 i skiftregisteret på 3 bit.

LFSR modulen med denne tilbakekoblingen vil genere samme tallsekvens som i forrige oppgave (dvs. 4,2,5,6,7,3,1, osv.) når skiftregisteret får start verdien ofte kalt «seed» lik 4 (dvs. binært «100»).

Forskjellige seed verdier skal kunne lastes inn i registeret med signalet seed når signalet load er aktivt høyt (dvs. '1').

Hvis signalet seed har verdien null som er en ulovlig verdi, skal et signal err bli aktivt høyt inntil en ny seed verdi blir lagt inn i skiftregisteret når load er aktivt høyt igjen.

Implementer arkitekturen til modulen LFSR som er vist under i syntetiserbar VHDL.

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

entity lfsr is
  port
    (rst   : in  std_logic;
     clk   : in  std_logic;
     load  : in  std_logic;
     seed  : in  std_logic_vector(2 downto 0);
     run   : in  std_logic;
     q     : out std_logic_vector(2 downto 0);
     err   : out std_logic);
end entity lfsr;

architecture rtl of lfsr is

  < Deklarer LFSR skiftregister her>

begin

  < Lag LFSR prosessen her >

end architecture rtl;
```

Oppgave 8 (8%)

I modulen compute vist under regnes det ut summen av tallene a, b, c og d hver på 16 bit. Utgangen result på 16 bit settes til max verdi lik X»FFFF» (dvs. alle bit satt til '1') når summen er større enn X»FFFF» og samtidig settes signalet max til '1'.

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

entity compute is
  port
    (rst    : in  std_logic;
     clk    : in  std_logic;
     a      : in  std_logic_vector(15 downto 0);
     b      : in  std_logic_vector(15 downto 0);
     c      : in  std_logic_vector(15 downto 0);
     d      : in  std_logic_vector(15 downto 0);
     result : out std_logic_vector(15 downto 0);
     max    : out std_logic);
end entity compute;

architecture rtl of compute is
begin

  process (rst, clk) is
    variable result_i : unsigned(17 downto 0);
  begin
    if rst = '1' then
      result <= (others => '0');
      max    <= '0';
    elsif rising_edge(clk) then
      result_i := unsigned("00" & a) + unsigned("00" & b) +
                  unsigned("00" & c) + unsigned("00" & d);
      if result_i > "00111111111111111" then
        result <= (others => '1');
        max    <= '1';
      else
        result <= std_logic_vector(result_i(15 downto 0));
        max    <= '0';
      end if;
    end if;
  end process;

end architecture rtl;
```

Det viser seg at det blir timing feil under under implementasjon i valgt teknologi og med den valgte klokkefrekvensen slik at arkitekturen rtl må endres til en ny arkitektur pipelined_rtl som bare har en addisjon (dvs. + operator) og en sammenligning (dvs. result_i>"00111111111111111") i hver klokkeperiode for å oppnå timing kravet. Det kan utføres flere addisjonsoperasjoner i parallell i hver klokkeperiode.

Implementer denne nye arkitekturen pipelined_rtl som vist under til compute modulen (dvs. samme entity) i syntetiserbar VHDL.

```
architecture pipelined_rtl of compute is
begin

  < Implementer VHDL koden her >

end architecture pipelined_rtl;
```

Oppgave 9 (3 %)

Hvor mange registere (dvs. flip-flip'er) har den oppgitte arkitekturen rtl i oppgave 8	A	15 registere	
	B	16 registere	
	C	17 registere	
	D	18 registere	
	E	19 registere	

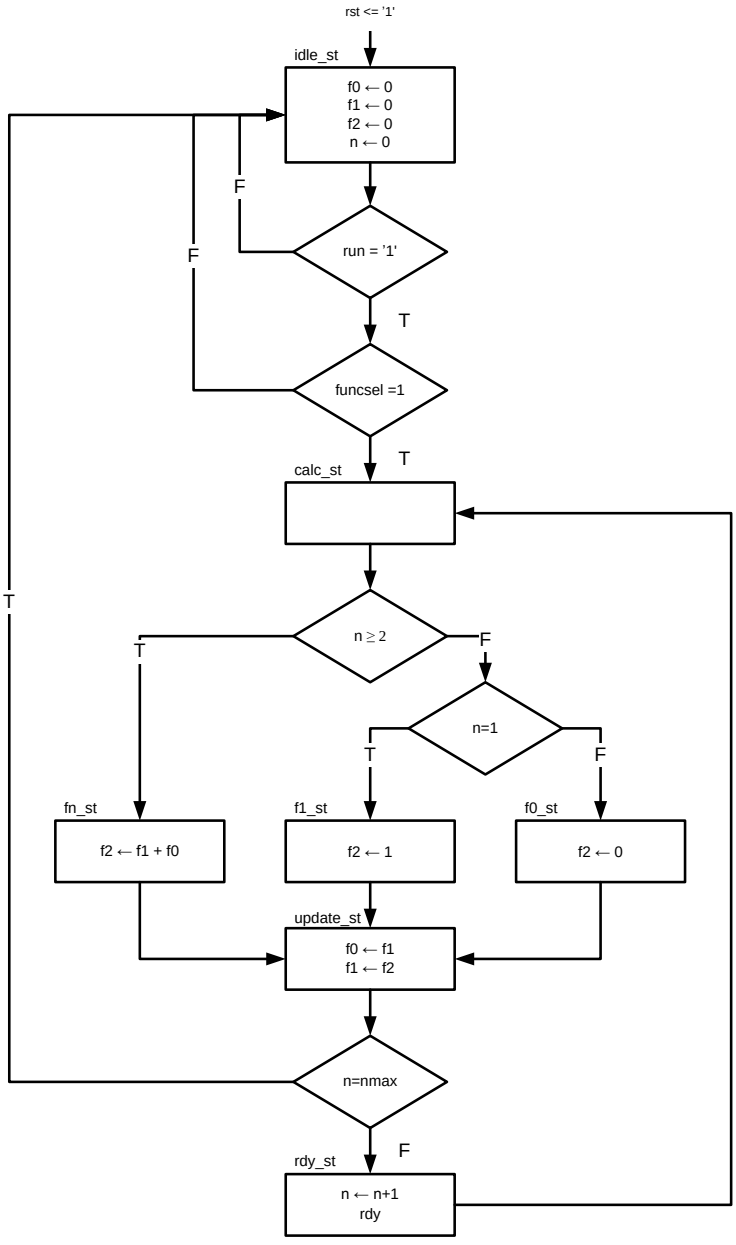
Oppgave 10 (4 %)

Hvor mange registere (dvs. flip-flip'er) har den pipelinede arkitekturen pipelined_rtl som ble implementert i oppgave 8?	A	51 registere	
	B	52 registere	
	C	53 registere	
	D	68 registere	
	E	69 registere	

Oppgave 11

ASM-diagrammet under viser en mulig implementasjon av en funksjon som genererer den velkjente Fibonacci-tallfølgen.

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases}$$



a) (15%)

Implementer ASM-diagrammet over som en to-process tilstandsmaskin i syntetiserbar (RTL) VHDL. Du må selv legge til flere interne signaler etter behov.
Fibonacci-tilstandsmaskinen har følgende entitet:

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

entity fibonacci is
  generic
    (
      FIBWIDTH : natural := 32 --antall bit i generert Fibonaccitall
      );
  port
    (
      rst       : in  std_logic; --asynkron reset
      clk       : in  std_logic; --clock
      run       : in  std_logic; --en run puls starter funksjonsgenratoren
      funcsel   : in  std_logic_vector(2 downto 0); --funksjonsvalg, "001"
                                                    --velger Fibonacci
      nmax      : in  std_logic_vector(7 downto 0); --angir max antall Fibonacci
                                                    --tall som skal genereres
      inum      : out std_logic_vector(7 downto 0); --angir Fibonaccitall index n
      rdy       : out std_logic; --en positiv puls med en clk-periodes
                                 --varighet for å angi et gyldig
      fn        : out std_logic_vector (FIBWIDTH-1 downto 0) --Fibonaccitall n
      );
end fibonacci;

architecture RTL_fibonacci of fibonacci is
  signal f0, next_f0 : unsigned(FIBWIDTH-1 downto 0);
  signal f1, next_f1 : unsigned(FIBWIDTH-1 downto 0);
  signal f2, next_f2 : unsigned(FIBWIDTH-1 downto 0);
  signal n, next_n   : unsigned(7 downto 0);

 <skriv flere deklarajoner her>

Begin

<skriv RTL-koden her>

--Concurrent statements
  inum <= std_logic_vector(n);
  fn   <= std_logic_vector(f2);

End architecture RTL_fibonacci;
```

b) (15%)

Lag en selvsjekkende testbenk i VHDL som sjekker de første 20 tallene som Fibonacci-tilstandsmaskinen over genererer.

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19
0	1	1	2	3	5	8	13	21	34	45	89	144	233	377	610	987	1597	2584	4181

Du kan anta at du har de første 20 (n=0-19) Fibonacci-tallene lagret i et internt signal i testbenken definert som:

```
constant MYNMAX : natural:= 20;
type fibfasit is array (0 to MYNMAX-1) of natural;
signal myfasit : fibfasit := (0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,
610,987,1597,2584,4181);
```

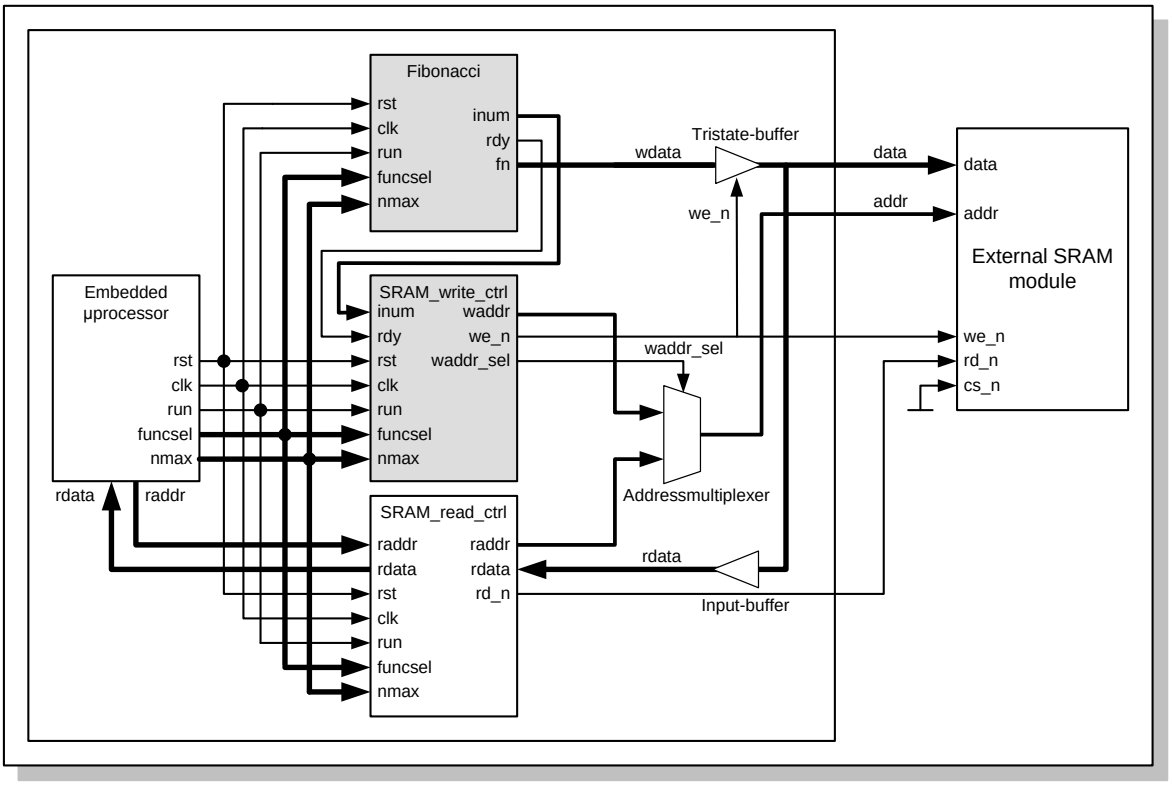
c) (6%)

Gjør om ASM-diagrammet til tilstandmaskinen beskrevet i a) til en Mealy-maskin. Dette skal redusere latency i tilstandsmaskinen med 2 klokkeperioder.

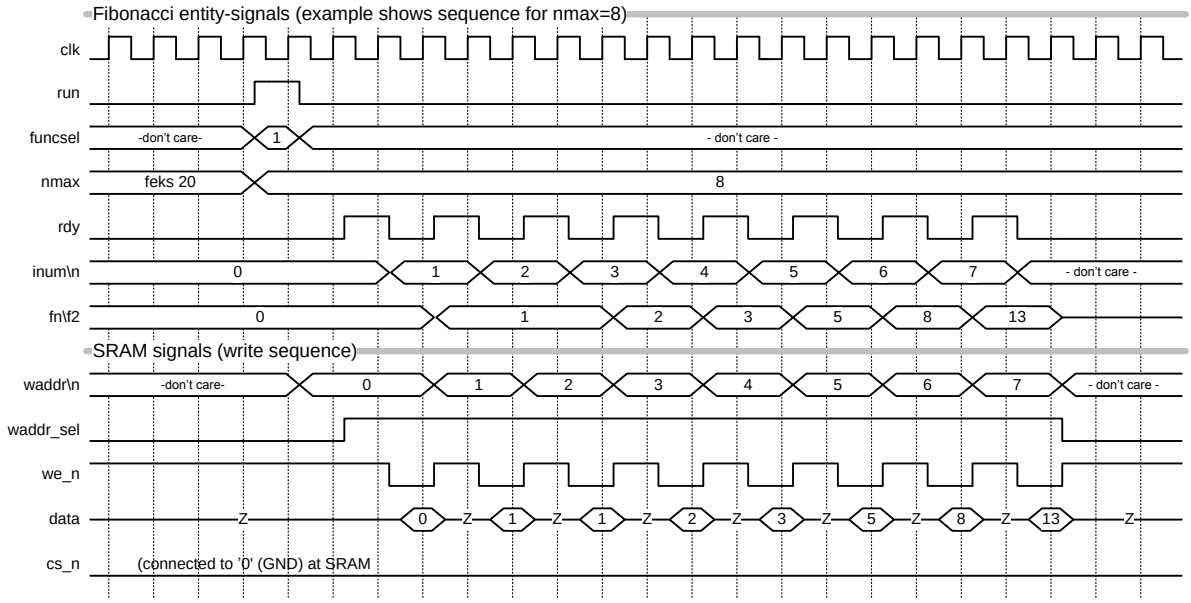
d) (10%)

Lag et ASM-diagram som en Mealy tilstandsmaskin for modulen SRAM_write_ctrl, se figuren under.
SRAM_write_ctrl's hovedoppgave er å lagre Fibonacci tallene forløpende i SRAM etter hvert som de blir generert.
Fibonacci-indeksnummeret ***inum,*** benyttes som skriveadresse.

Fibonacci, SRAM_write_ctrl og SRAM_read_ctrl (skal ikke implementeres) er en del av et tenkt system for å lage en del optimaliserte funksjoner i et embedded microprosessorsystem.



Signalene ***rd_n***, ***we_n*** og ***cs_n*** er alle aktivt lave.



Legg merke til at timingdiagrammet over følger tilstandsmaskinen som ble beskrevet i oppgave 11c)

Virkemåten til SRAM_write_crtl kan summeres opp punktvis:

1. Skrive-funksjonen starter ved at SRAM_write_crtl sjekker at inputsignalene ***run*** og ***funcsel*** er henholdsvis '1' og «001». Deretter settes ***waddr_sel*** aktiv '1'. ***waddr_sel*** styrer adressemultiplekseren slik at ***waddr*** blir aktiv adresse til SRAM (se figuren på forrige side). Det er viktig at ***waddr_sel*** er aktivt helt til alle Fibonacci-tallene er lagret.
2. Deretter venter den på at ***rdy***-signalet skal gå aktivt høyt, som en indikasjon på at et nytt Fibonacci-tall er klart til å lagres.
3. I klokkeperioden etter at ***rdy*** har gått aktivt høyt skal ***we_n*** gå aktivt lavt i en klokkeperiode. Data blir lagret i adressen ***waddr(addr)*** på stigende flanke av ***we_n.*** I dette tilstandskiftet får skriveadressen ***waddr***, verdien til ***inum*** (dvs. neste skriveadresse, se timingdiagram).
4. Punkt 2) og 3) repeteres helt til alle Fibonacci-tallene er lagret, dvs. ***nmax*** antall skriveoperasjoner.

e) (6%)

Lag en process som beskriver tristate-bufferet i figuren på forrige side ved å benytte ***wdata*** og ***we_n*** vist i figuren som inputs.

Vedlegg 1.
INF3430/INF4431. Oppgavesvar for kandidat nr: _____

Oppgave	A	B	C	D	E
1					
2					
3					
4					
5					
9					
10					