INF3430

Funksjoner og prosedyrer
Standardbiblioteker
Komplekse sekvensielle systemer

Innhold

- Funksjoner og operatorer
- Prosedyrer
- Begrepet overload
- Biblioteker
  - Package/package body
- Standard biblioteker
- Komplekse sekvensielle systemer
  - Eksempel på en enkel mikroprosessor

Funksjoner (1)

- Benyttes mye i modeller for simulering
  - Høynivåbeskrivelser for
    - Beregninger
    - type konvertering
- Funksjoner som har anvendelser for syntese inkluderer
  - Type konvertering
  - Bruk av funksjoner som alternativ til komponent instansiering
  - Overload operatorer og funksjoner

- Mange nyttige funksjoner definert i standard pakker (packages)
  - IEEE 1076
  - IEEE 1164 (std_logic_1164)
  - Synopsys'biblioteker (støttet av de fleste syntheseverktøy)
  - IEEE 1076.3 (Syntese standard)

Funksjoner (2)

- Funksjons beskrivelse
  - Kan bare ha input signaler som parametre
  - Returner kun èn verdi (dette kan være et signal eller en vector)
  - Sekvensiell beskrivelse
  - Kan ikke ha ”wait statement”
  - Kan ikke definere signaler internt

```
Architecture func of functest is

-- Deklarasjoner
function bl2bit(a: BOOLEAN)  return bit is
begin
…
end bl2bit;
…
-- Start of architecture
begin
…
end func;
```

- Funksjons deklarasjon
  - en funksjon deklareres i den deklarative regionen av en arkitektur
  - eller den deklarative regionen av en process

Funksjoner (3)

- Funksjons deklarasjoner
  - Kan også deklareres i et "package/package body"par.
- Et funksjonsbibliotek er bygd opp på denne måten
- I packagedelen kan man også legge inn:
  - Komponentdeklarasjoner
  - Datatypedefinisjoner
  - Konstanter

```
package my_func is
  function bl2bit(a:BOOLEAN) return bit;
  .. Flere funksjoner
end my_func;

--  Package body
package body my_func is
  function bl2bit(a:BOOLEAN) return bit is
  begin
    if a then
      return '1';
    else
      return '0';
    end if;
  end bl2bit;
--  Flere funksjoner
….
end my_func;
```

Operatorer

- Operatorer defineres på samme måte som funksjoner, men ved: “<operatornavn>”
- Operatorer benyttes forskjellig fra funksjoner:

```
function "+" (a,b :std_logic_vector)
          return std_logic_vector;
..
sum <= a + b;
-----
function add (a,b):std_logic_vector)
          return std_logic_vector;
..
sum <= add(a,b);
```

Bruk av funksjonsbibliotek

- Et “package/package body”par kan være kompilert til work

```
use work.my_func.all;
-- eller
-- use.work.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig

entity ......
architecture ...
signal a: BOOLEAN;
signal b: bit;
begin
  b <= bl2bit(a);
end architecture ..;
```

- eller til et annet bibliotek. Dette må ha et *logisk navn*. Vi antar mylib.
- Det logiske navnet gis i det aktuelle verktøyet og gjenspeiles gjerne i lagerstrukturen til verktøyet

```
library mylib;
use my_lib.my_func.all;
-- eller
-- use.my_lib.my_func.bl2bit;
-- dersom bare bl2bit skal gjøres synlig
```

Overloading (1)

- Overloading betyr å definere samme operator, funksjon eller prosedyre for forskjellige datatyper eller blanding av datatyper.
- De kan være forskjellig antall parametre i de forskjellige overload operatorene, funksjonene eller prosedyrene.
- VHDL-analyse/syntese programmet skiller de forskjellige overload operatorene, funksjonene eller prosedyrene fra hverandre ved å se på datatypen til de aktuelle parametrene og sammenligner dem med de formelle parametrene.

Overloading (2)

- Det finnes en rekke standard biblioteker med overload operatorer, funksjoner og prosedyrer i IEEE 1164 og 1076.3
  - IEEE 1164
    - Package std_logic_1164
    - Synopsys biblioteker (kompilert til IEEE)
      - Package std_logic_unsigned
      - Package std_logic_signed
      - Package std_logic_arith
  - IEEE1076.3
    - Package numeric_std
- NB! Det er viktig å sjekke ut hva det aktuelle syntheseverktøyet støtter.
- De forskjellige syntheseverktøyene kan ha organisert disse bibliotekene forskjellig.
  - Det betyr at man må endre “library use clause” i VHDL-filen som skal syntetiseres.

Prosedyrer

- Kan returnere flere enn èn verdi
- Retur skjer via prosedyreparametrene
  - Parametre som skal returneres må være deklarert som mode out eller inout.
- Kun sekvensiell konstruksjon
- Kan ha wait statement
- Størrelser på vektorer defineres av aktuelle parametre
- Deklareres på samme steder som funksjoner og operatorer

```
procedure dff ( signal d: std_logic_vector;
                signal clk, rst: std_logic;
                signal q: out std_logic_vector) is
begin
  if rst = '1' then
    for i in d'range loop
      q(i) <= '0';
    end loop;
  elsif clk'event and clk = '1' then
    q <= d;
  end if;
end dff;
```

Nyttige prosedyrer i testbenker

- Pakken *std_textio* fra IEEE og *std_logic_textio* fra Synopsys inneholder en rekke nyttige prosedyrer for å lese fra fil og skrive til fil:
  - Bits og vectorer(som enkeltbit)
    - read
    - write
  - Octalt format
    - oread
    - owrite
  - Hexadesimalt format
    - hread
    - hwrite

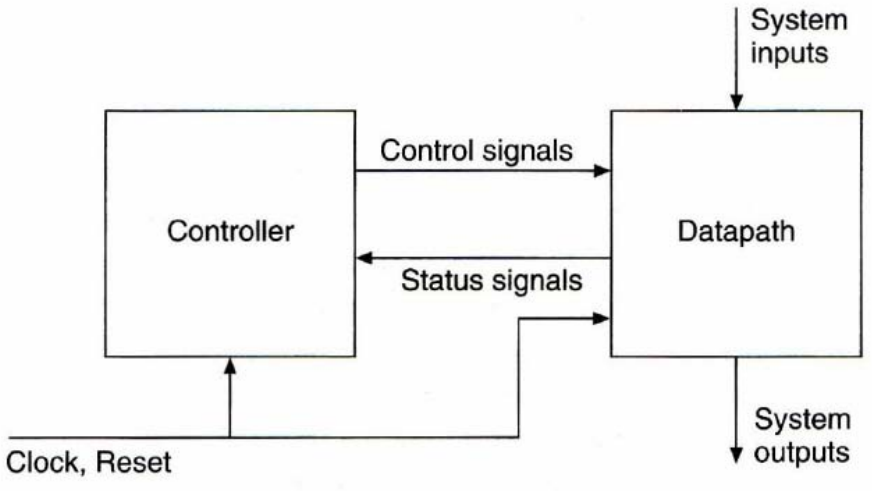
  Disse finnes i en rekke overloadede utgaver

Nyttige prosedyrer i testbenker(2)

- Biblioteket Std_developers_kit som følger med Modelsim inneholder en rekke nyttige pakker.
  - Spesielt nyttig i testbenker er pakken STD_IOPAK.
  - STD_IOPAK inneholder en rekke funksjoner for å manipulere strenger og fil I/O.

Komplekse sekvensielle systemer

- Vanlig å dele opp et sekvensielt system i to deler:
  - Datapath (datavei)
  - Controller
- Datapath'en gir strukturen og virkemåten til systemet, mens
- Kontrolleren kontrollerer virkemåten og timingen til datapath operasjonene
- Start alltid med å tegne datapath fordi da er det gitt hva som skal kontrolleres



Eksempel:En enkel mikroprosessor

- En program i en mikroprossor består av en rekke assemblyinstruksjoner.
- For eksempel kan man ha følgende c-kode:

  a=b+c; som kompileres til assemblyinstruksjonene:

  LOAD b

  ADD c

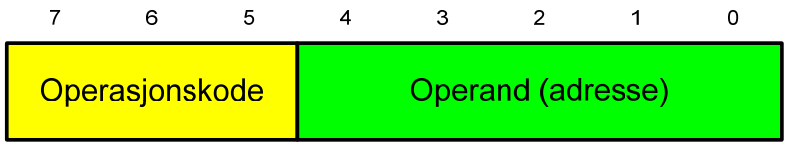
  STORE a

  – En assemblyinstruksjon består av en operasjonskode og en operand. Operanden tilsvarer en adresse

Instruksjoner

- I vårt eksempel tenker vi oss en 8-bits instruksjon med en operasjonskode på 3 bit og en operand på 5.
- Det betyr at vi kan ha 8 forskjellige operasjonskoder og 32 adresser
- Assemblyprogrammet over kan oversettes til følgende maskinkode:

LOAD b	00000001
ADD c	01000010
STORE a	00100011



Adresseringsmodi

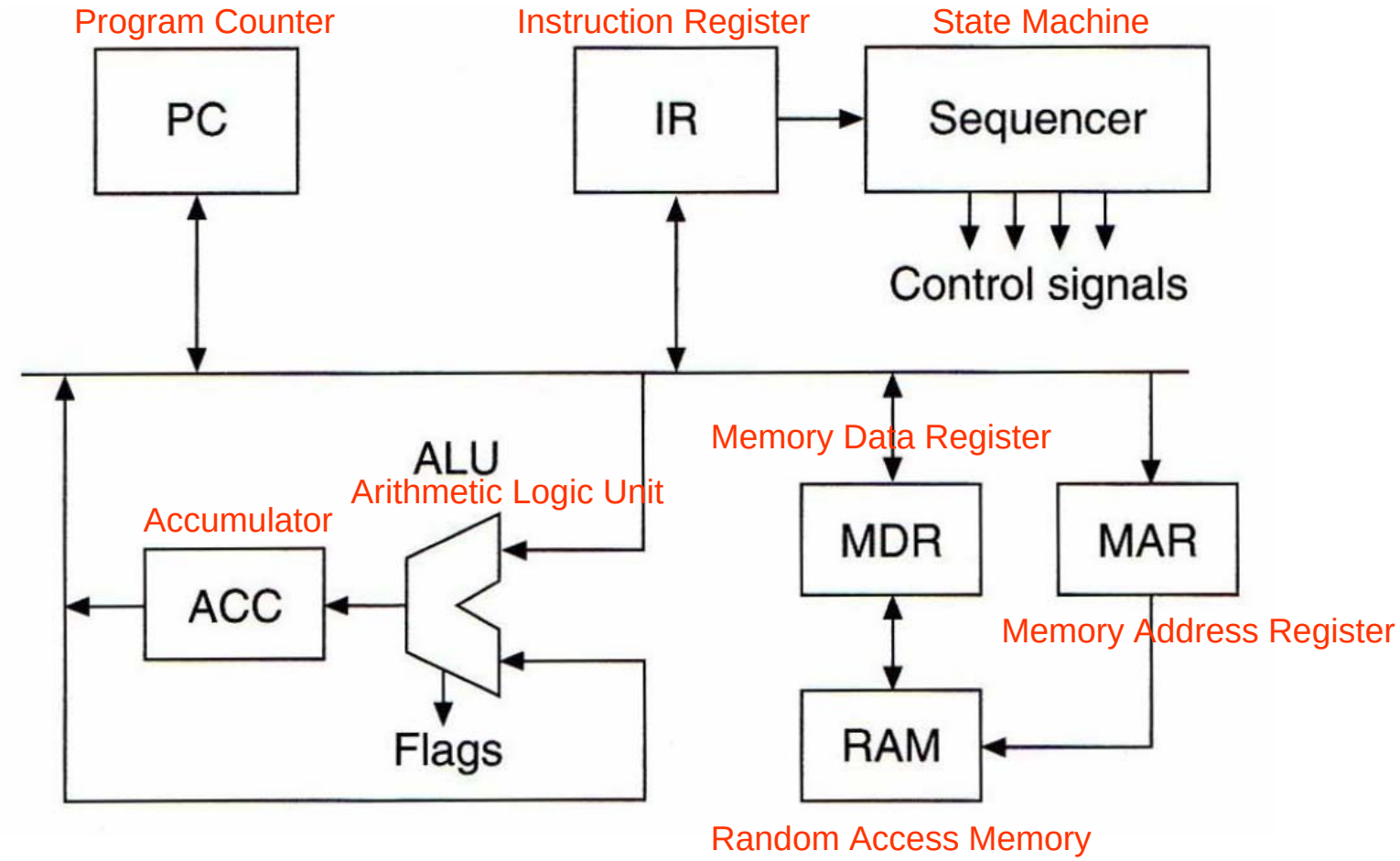
- Vår mikroprosessor støtter ”direct mode” adressering
  - Det betyr at data som skal opereres på må tas fra adressen gitt av operanden
- For eksempel vil ikke mikroprosessoren vår støtte instruksjoner av typen

  LOAD b

  ADD 5

  STORE
  - Her er ADD 5 eksempel på immediate mode adressering. Det betyr at instruksjonen opererer direkte på operanden.
  - Direct og immediate mode instruksjoner krever forskjellig datapath og er å betrakte som vidt forskjellige instruksjoner.

Kontroll og dataveier



Kontroll og dataveier

- PC-Program Counter
  - Inneholder adressen til neste instruksjon
- IR-Instruction Register
  - Inneholder operasjonskoden av instruksjonen og er input til
- SEQUENCER
  - Tilstandsmaskin som dekoder operasjonskoden og lager kontrollsignaler til det enkelte delene av mikroprosessoren
- RAM
  - Inneholder instruksjoner og data
  - MDR og MAR er synkroniseringsregistre til RAM (som er asynkron i dette tilfellet)
- ALU-Arithmetic Logic Unit
  - Utfører aritmetiske/logiske operasjoner
  - Mikroprosessorens hjerne
- ACC-Accumulator
  - Lagrer resultat av en ALU operasjon

Kontrollsignaler

Table 7.1 Control signals of microprocessor.

ACC_bus	Drive bus with contents of ACC (enable three-state output)
load_ACC	Load ACC from bus
PC_bus	Drive bus with contents of PC
load_IR	Load IR from bus
load_MAR	Load MAR from bus
MDR_bus	Drive bus with contents of MDR
load_MDR	Load MDR from bus
ALU_ACC	Load ACC with result from ALU
INC_PC	Increment PC and save the result in PC
Addr_bus	Drive bus with operand part of instruction held in IR
CS	Chip Select. Use contents of MAR to set up memory address
R_NW	Read, Not Write. When false, contents of MDR are stored in memory
ALU_add	Perform an add operation in the ALU
ALU_sub	Perform a subtract operation in the ALU

PC-Program Counter

```
sysbus <= rfill & std_logic_vector(count) when PC_bus = '1' else
          (others => 'Z');

process (clock, reset) is
begin
  if reset = '1' then
    count <= (others => '0');
  elsif rising_edge(clock) then
    if load_PC = '1' then
      if INC_PC = '1' then
        count <= count + 1;
      else
        count <= unsigned(sysbus(word_w - op_w - 1 downto 0));
      end if;
    end if;
  end if;
end process;
```

IR-Instruksjonsregister

```
sysbus <= rfill & instr_reg(word_w - op_w - 1 downto 0) when
          Addr_bus = '1' else (others => 'Z');
op <= slv2op(instr_reg(word_w - 1 downto word_w - op_w));

process (clock, reset) is
begin
  if reset = '1' then
    instr_reg <= (others => '0');
  elsif rising_edge(clock) then
    if load_IR = '1' then
      instr_reg <= sysbus;
    end if;
  end if;
end process;
```

MAR, MDR og RAM

```
begin
  if reset = '1' then
    mdr <= (others => '0');
    mar <= (others => '0');
    mem := prog;
  elsif rising_edge(clock) then
    if load_MAR = '1' then
      mar <= unsigned(sysbus(word_w - op_w - 1 downto 0));
    elsif load_MDR = '1' then
      mdr <= sysbus;
    elsif CS = '1' then
      if R_NW = '1' then
        mdr <= mem(to_integer(mar));
      else
        mem(to_integer(mar)) := mdr;
      end if;
    end if;
  end if;
end process;
```

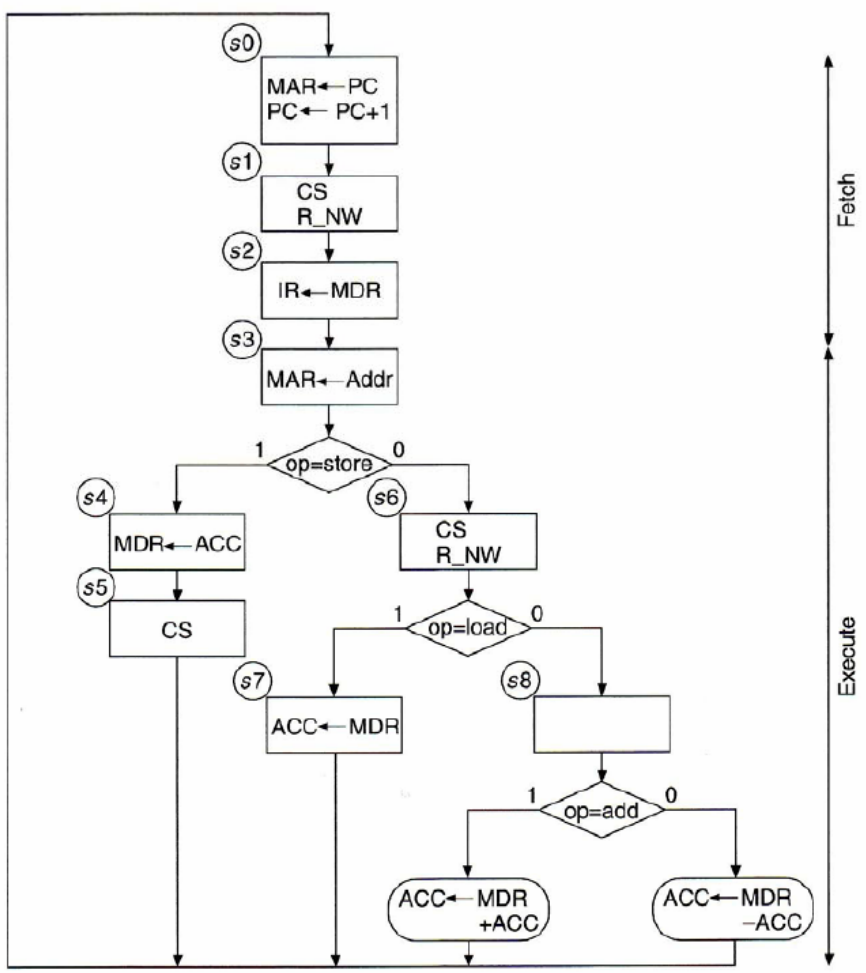
ALU-Aritmetisk Logisk Enhet

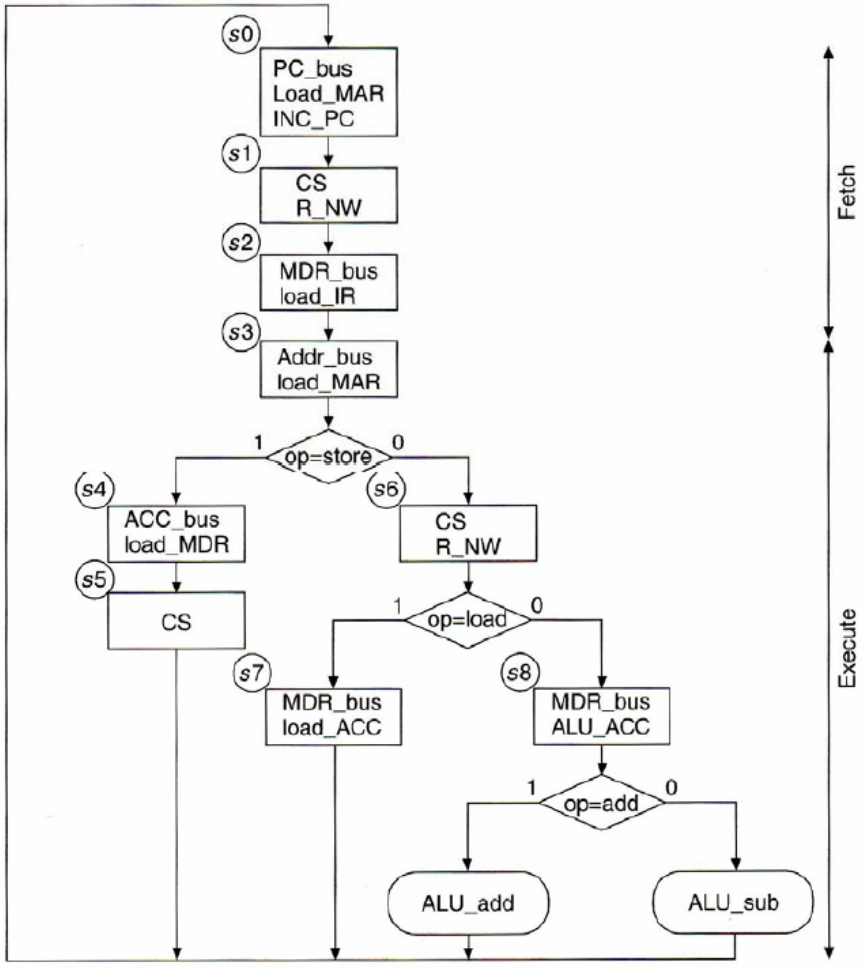
```
sysbus <= std_logic_vector(acc) when ACC_bus = '1' else
          (others => 'Z');
z_flag <= '1' when acc = zero else '0';

process (clock, reset) is
begin
  if reset = '1' then
    acc <= (others => '0');
  elsif rising_edge(clock) then
    if load_ACC = '1' then
      if ALU_ACC = '1' then
        if ALU_add = '1' then
          acc <= acc + unsigned(sysbus);
        elsif ALU_sub = '1' then
          acc <= acc - unsigned(sysbus);
        end if;
      else
        acc <= unsigned(sysbus);
      end if;
    end if;
  end if;
end process;
```

Instruksjonssett

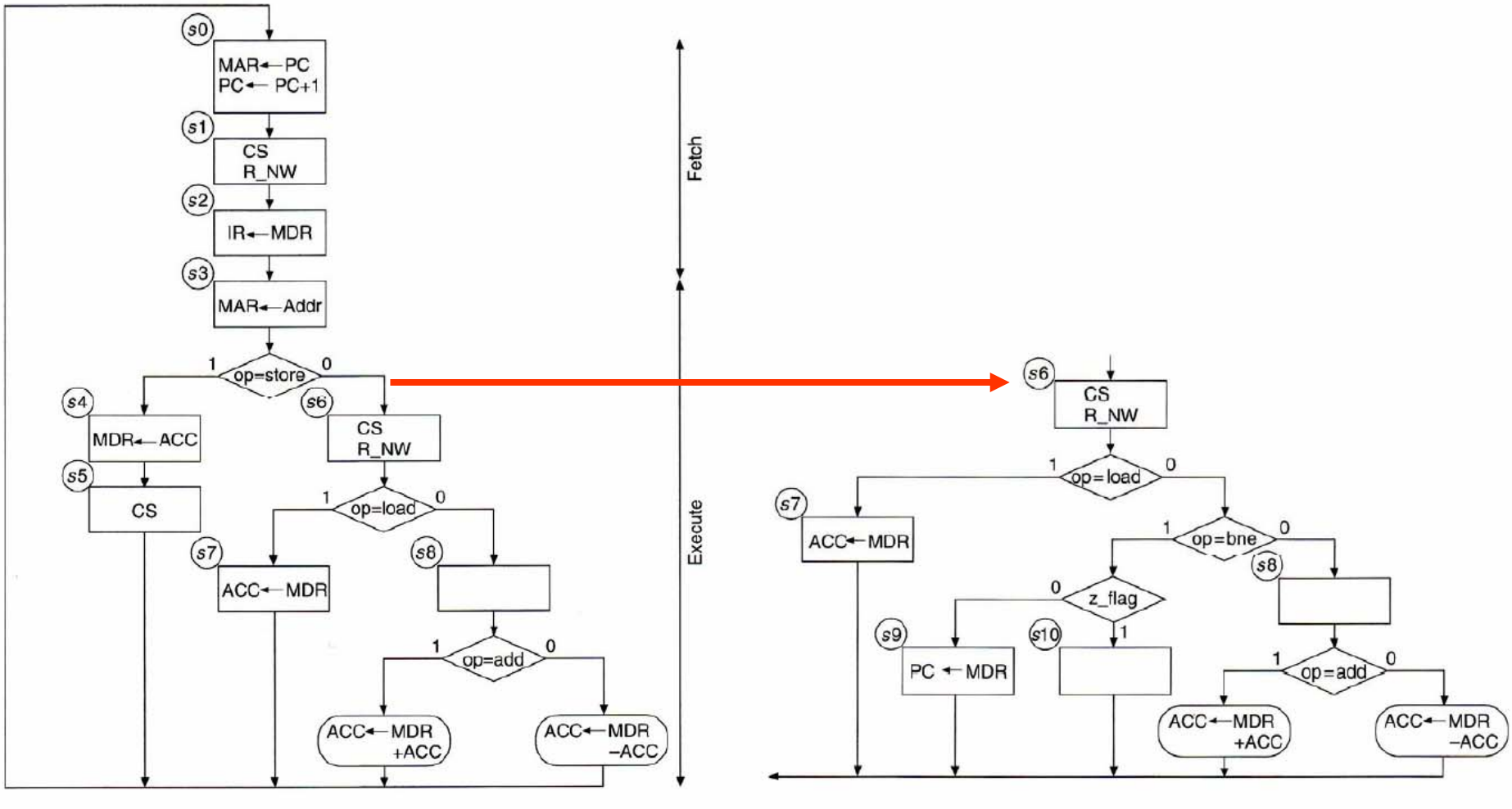
- LOAD
- STORE
- ADD
- SUB
- BNE

ASM flytdiagram (1)





ASM flytdiagram (2)



Biblioteker (1)

- Pakken cpu_defs

```
library IEEE;
use IEEE.std_logic_1164.all;
package cpu_defs is

  type opcode is (load, store, add, sub, bne);
  constant word_w       : NATURAL := 8;  -- no. of bits for bus
  constant op_w         : NATURAL := 3;    -- no. of bits for opcode
  constant rfill        : std_logic_vector(op_w - 1 downto 0)
                        := (others => '0'); -- padding for address
  function slv2op (slv  : in std_logic_vector) return opcode;
  function op2slv (op   : in opcode) return std_logic_vector;
end package cpu_defs;
```

Biblioteker (2)

- Dersom mikroprosessoren skal syntetiseres må operasjonskoden konvertereres til std_logic_vector eller lignende
  - Typekonverteringsfunksjonene ***op2slv*** og ***slv2op***

```
package body cpu_defs is

  type optable is array (opcode)
                  of std_logic_vector(op_w - 1 downto 0);
  constant trans_table : optable
           := ("000", "001", "010", "011", "100");

  function op2slv (op : in opcode) return std_logic_vector is
    begin
    return trans_table(op);
  end function op2slv;
```

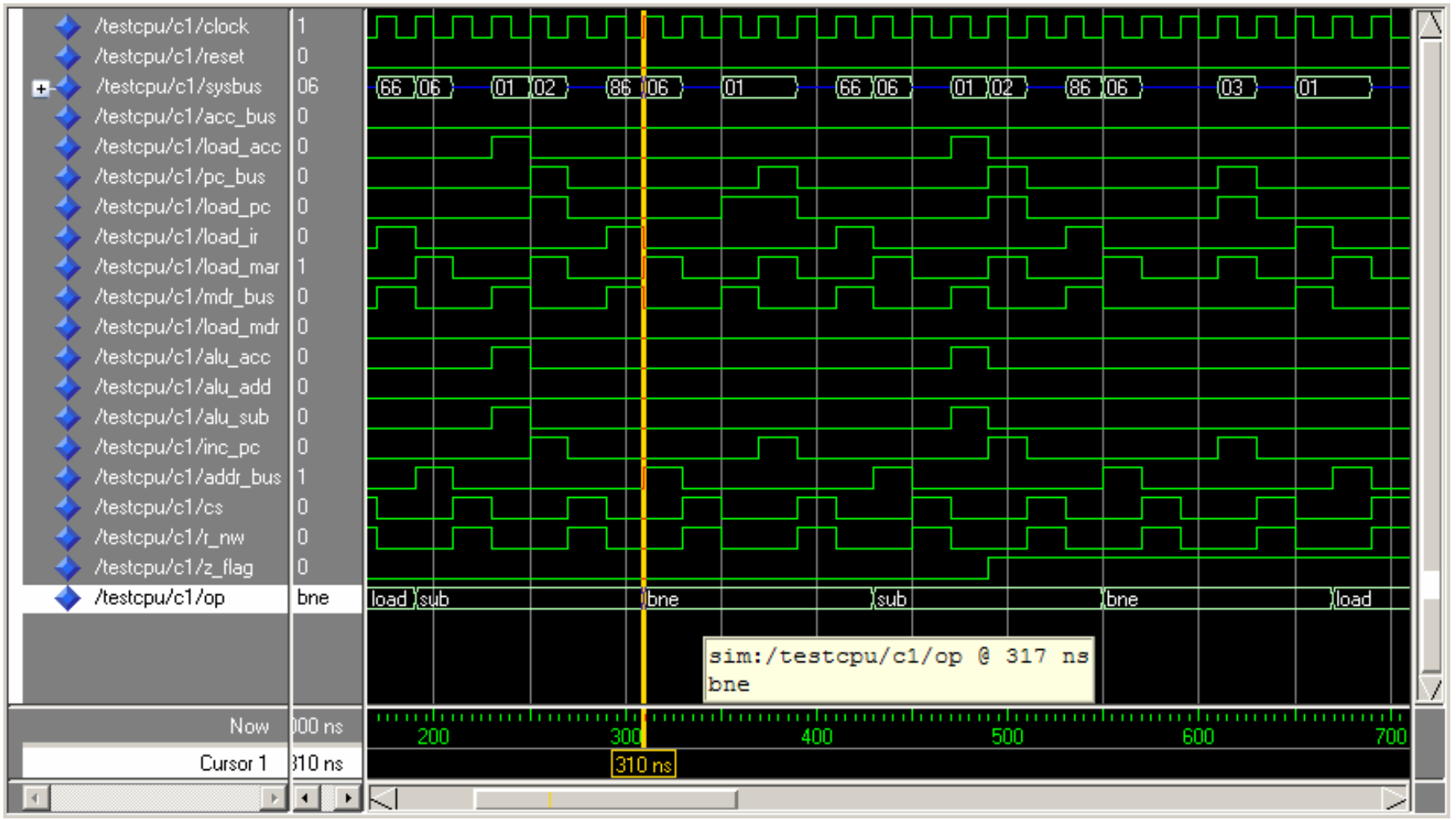
Biblioteker (3)

```
  function slv2op (slv : in std_logic_vector) return opcode is
    variable transop : opcode;
  begin
-- This is how it should be done, but some synthesis tools may not
-- support this.
    for i in opcode loop
      if slv = trans_table(i) then
        transop := i;
      end if;
    end loop;
-- This is a less elegant method!
-- If the definitions of opcode and/or trans_table are changed, this
-- code also has to be changed. There is therefore potential for
-- inconsistency.
--case slv is
--   when "000" => transop := load;
--   when "001" => transop := store;
--   when "010" => transop := add;
--   when "011" => transop := sub;
--   when "100" => transop := bne;
--end case;
    return transop;
  end function slv2op;

end package body cpu_defs;
```

Programmet

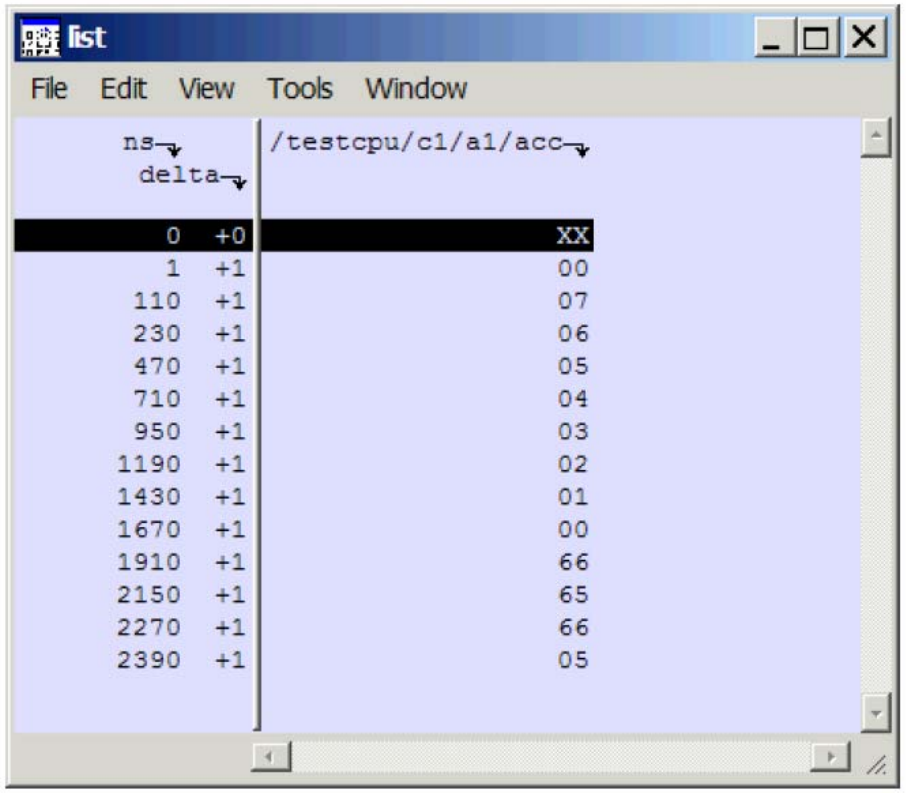
```
process (clock, reset) is
  type mem_array is array (0 to 2**(word_w - op_w) - 1) of
                      std_logic_vector(word_w - 1 downto 0);
  variable mem : mem_array;
  constant prog : mem_array := (
    0 => op2slv(load)  & std_logic_vector(to_unsigned(5, word_w - op_w)),
    1 => op2slv(sub)   & std_logic_vector(to_unsigned(6, word_w - op_w)),
    2 => op2slv(bne)   & std_logic_vector(to_unsigned(6, word_w - op_w)),
    3 => op2slv(load)  & std_logic_vector(to_unsigned(1, word_w - op_w)),
    4 => op2slv(store) & std_logic_vector(to_unsigned(5, word_w - op_w)),
    5 => std_logic_vector(to_unsigned(2, word_w)),
    6 => std_logic_vector(to_unsigned(1, word_w)),
    others => (others => '0'));

begin
```

Modelsim. Utdrag av timingdiagram



Modelsim. “List” vindu

- Bruk ”list”-vindu til å se på kun endringer til et signal



Modelsim. Memory vinduet

- Bruk av ”memory”vindu