Programmerbara kretsar och VHDL 2

Föreläsning 10

Digitalteknik, TSEA22

Oscar Gustafsson
Anders Nilsson
Institutionen för systemteknik

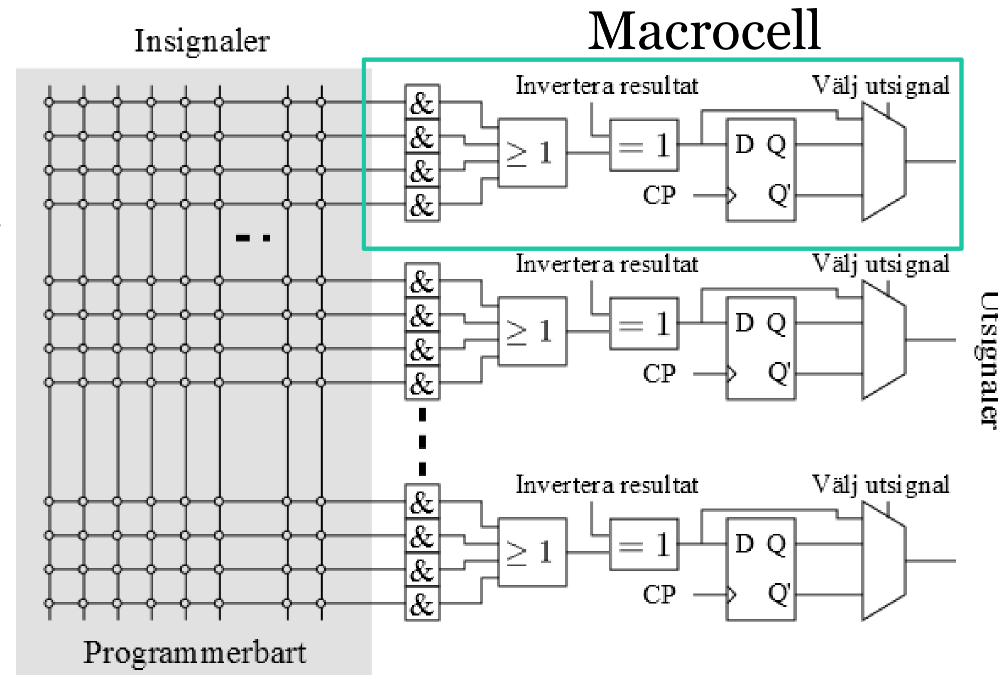
LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Programmerbara kretsar igen
- Mer om processer
- Egna typer
- Använda byggblock
- Generella byggblock
- Lab 4

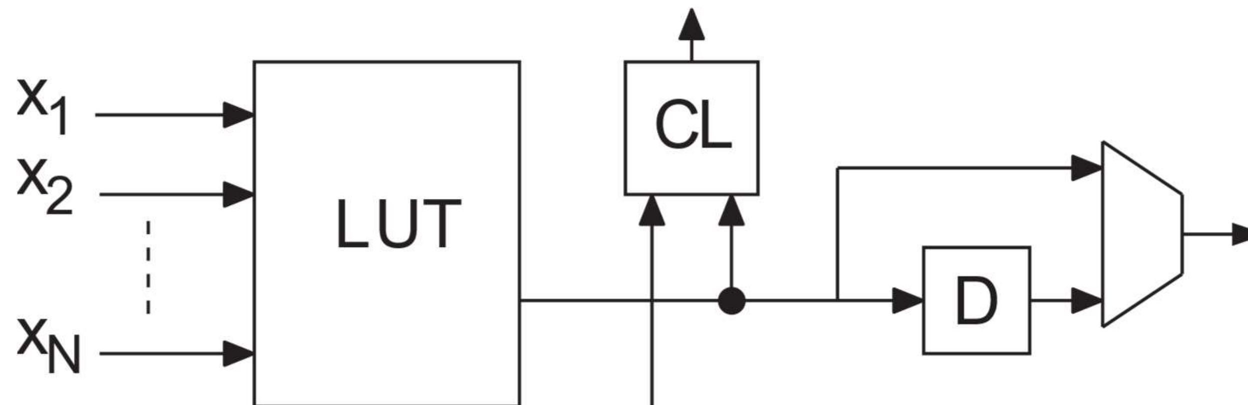
Programmerbara kretsar

- CPLD
  - Ett lager av summa-av-produkter
- Finns det någon annan typ av generell struktur?



FPGA

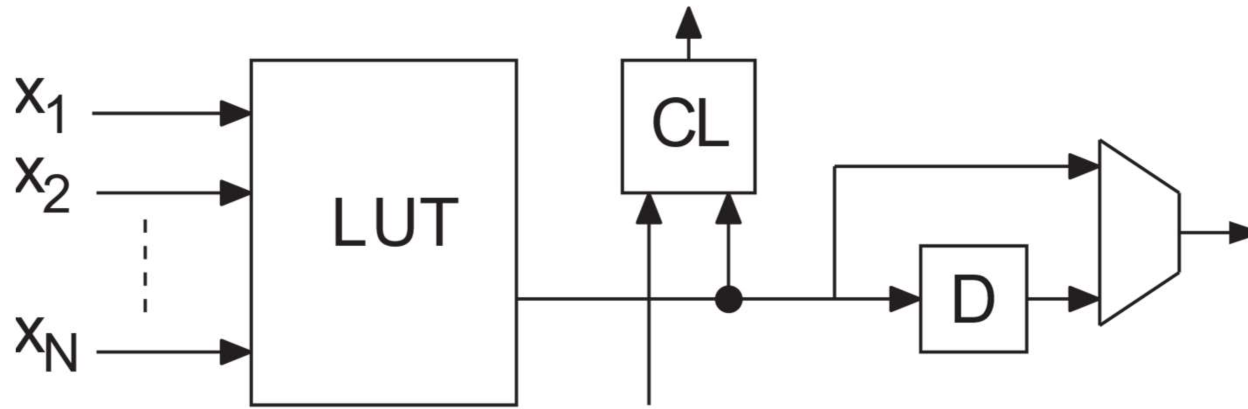
- PROM/tabell/minne går att använda till alla kombinatoriska kretsar
- FPGA – programmerbar krets med en massa tabeller



FPGA

- Innehåller ofta dessutom
  - Större minnen
  - Multiplikatorer/DSP block
- Kan innehålla processorer

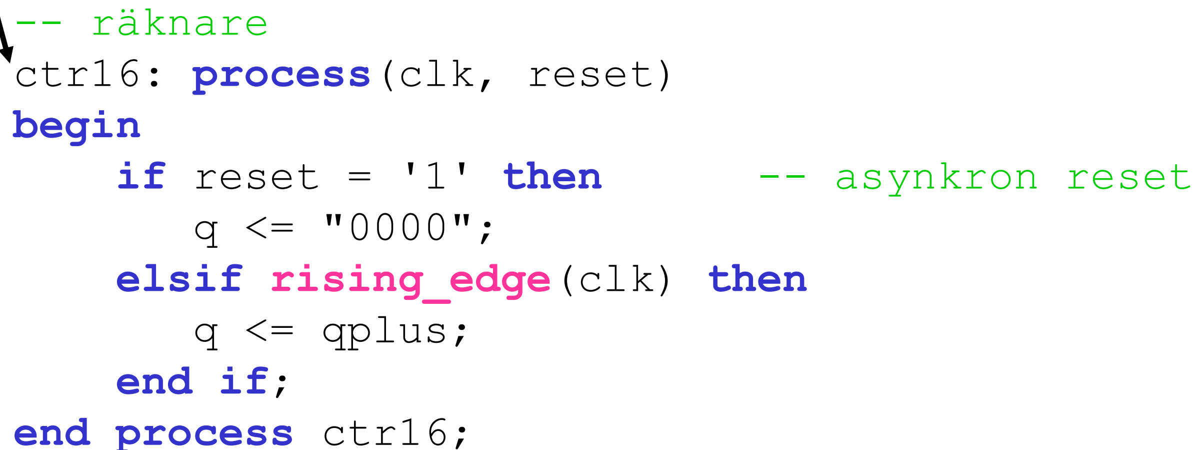
FPGA



- LUT – look-up table
- CL – carry-logic för adderare
- Kallas CLB eller ALM

Mer om processer

- Processer kan döpas
  - Användbart om det finns flera processer
  - Syns i simulatorn (går att leta reda på processer)



Mer om processer

- Det går göra mer än bara vippor med processer
- Ett exempel som kan vara användbart är **`case-when`** konstruktionen

```
case (styrsignal) is
  when (värde 1)   => (sats 1);
  when (värde 2)   => (sats 2);
   ...
  when (värde n-1) => (sats n-1);
  when others      => (sats n);
end case;
```

Mer om processer

- **`case-when`** motsvarar **`with-select-when`**
- En skillnad är att satserna kan vara godtyckligt stora, dvs inte bara en tilldelning

```
case (styrsignal) is
  when (värde 1)   => (sats 1);
  when (värde 2)   => (sats 2);
   ...
  when (värde n-1) => (sats n-1);
  when others      => (sats n);
end case;
```

Programsatserna `case-when` och `if-then` får bara användas inne i en process.

Programsatserna `with-select-when` och `when-else` får bara användas utanför en process.

Mer om processer

- Det går att göra rent kombinatoriska processer

```
process(b, c)
begin
  a <= b and c;
end process;
```

- Är helt ekvivalent med

```
a <= b and c;
```

- Möjlig fördel: **`if-then-else`** och **`case-when`**
- Uppenbar nackdel: det finns ett flertal sätt att göra fel med rent kombinatoriska processer

Vanliga fel i kombinatoriska processer

- Fattas en signal i känslighetslistan

```
process(b)
begin
  a <= b and c;  -- a ändras inte om bara c ändras
end process;
```

- Utsignalen tilldelas inte alltid

```
process(b, c)
begin
  if c = '1' then
    a <= b;      -- a ändras inte då c blir 0
  end if;
end process;
```

Vanliga fel i kombinatoriska processer

- Simulatorn kommer göra exakt det som står
- Syntesverktyget kommer försöka göra så gott det kan för att få exakt det beteende som beskrivits
  - T ex hålla kvar värden med hjälp av latchar, vilket ni inte vill eftersom det var en och-grind som önskades

Använd bara kombinatoriska processer om du vet att du behöver det!

Mer om processer

- Tilldelningen av nya signalvärden sker när processen lämnas
- Hårdvaran ska bete sig på samma sätt som processen
- Några konsekvenser av detta är:
  - En signal kan tilldelas flera gånger i processen
  - Ett nytt tilldelat värde kan inte användas senare i processen, utan först vid "nästa varv"
  - Alla signaler som tilldelas innanför `if rising_edge` har en vippa på sig

Mer om processer

- En signal kan tilldelas flera gånger i processen

```
process(clk)
begin
   if rising_edge(clk) then
      a <= b;   -- a+ = b
      if c > 2:
        a <= c; -- a+ = c
      end if;
    end if;
end process;    -- a = a+
```

fungerar utan problem

Jämför

```
process(b, c, d)
begin
  a <= b and c;   -- a+ =…
  if d = '1':
    a <= b or c; -- a+ =…
  end if;
end process;      -- a = a+
```

I processen blir det egentligen bara en tilldelning av **a**, dvs när processen avslutas.

```
-- kombinatoriskt
a <= b and c;
a <= b or c when d = '1';
```

Går inte!

Två grindar driver samma signal!

(Dessutom med latchar)

Flera tilldelningar är oftast ett tecken på dålig kod

```
process(b, c, d)
begin
  a <= b and c;
  if d = '1':
    a <= b or c ;
  end if;
end process;
```

```
process(b, c, d)
begin
  if d = '1':
    a <= b or c ;
  else
    a <= b and c;
  end if;
end process;
```

Sämre kod!

Bättre kod!

Det är att föredra att en variabel bara tilldelas en gång

Mer om processer

- Tilldela inte samma signal i flera olika processer

```
process(b, c, d)
begin
     if d = '0' then
        a <= b and c;
     end if;
end process;
```

```
process(b, c, d)
begin
     if d = '1' then
        a <= b or c;
     end if;
end process;
```

fungerar ej!

Jämför

```
process(b, c, d)
begin
   if d = '0' then
       a <= b and c;
   end if;
end process;
```

```
process(b, c, d)
begin
   if d = '1' then
      a <= b or c;
   end if;
end process;
```

```
-- kombinatoriskt
a <= b and c when d = '0';
a <= b or c when d = '1';
```

I bägge fallen:

Två grindar driver samma signal!
(Dessutom med latchar)

```
-- korrekt
a <= b and c when d = '0'
else b or c;
```

Mer om processer

- Själva tilldelningen sker i slutet av processen

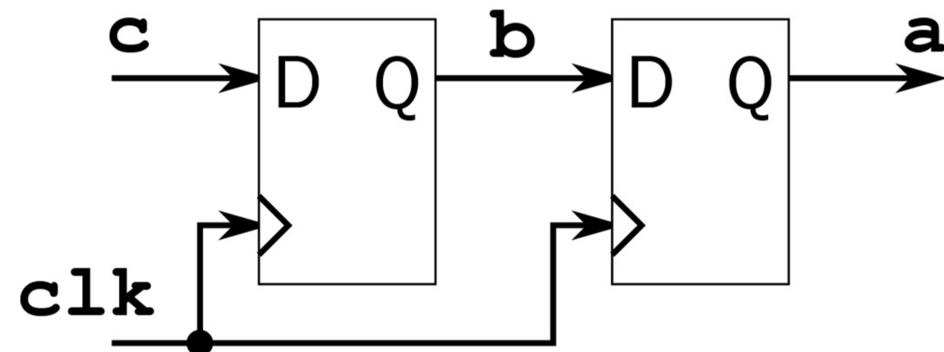
```
process(clk)
begin
   if rising_edge(clk) then
      b <= c;   -- b+ <= c
      a <= b;   -- a+ <= b
    end if;
end process;    -- a <= a+, b <= b+
```

- Om `b = '0'` och `c = '1'` när klockflanken kommer, vad har `a` och `b` för värden efter processen?
- `a = '0'` och `b = '1'`, för `b` blir inte `'1'` förrän när processen lämnas

Mer om processer

– Alla signaler som tilldelas innanför `if rising_edge` har en vippa på sig

```
process(clk)
begin
   if rising_edge(clk) then
      b <= c;
      a <= b;
   end if;
end process;
```



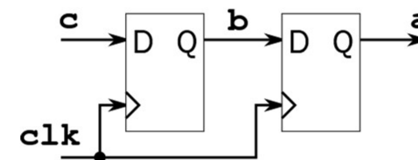
Tre ekvivalenta sätt att beskriva två vippor


```
process(clk)
begin
  if rising_edge(clk) then
    b <= c;   -- b+ = c
    a <= b;   -- a+ = b
  end if;
end process; -- a=a+,b=b+
```

```
process(clk)
begin
  if rising_edge(clk) then
    a <= b;   -- a+ = b
    b <= c;   -- b+ = c
  end if;
end process; -- a=a+,b=b+
```

```
process(clk)
begin
  if rising_edge(clk) then
    b <= c;   -- b+ = c
  end if;
end process; -- b=b+

process(clk)
begin
  if rising_edge(clk) then
    a <= b;   -- a+ = b
  end if;
end process; -- a=a+
```



Mer om processer

```
process(clk)
begin
  if rising_edge(clk) then
    count <= count + 1;   -- count+ = count+1
    if count = 8 then
      count <= "0000";    -- count+ = "0000"
    end if;
  end if;
end process;              -- count=count+
```

Hur långt räknar den?

Här kommer **count** att vara 8 i en klockcykel innan den blir 0 i nästa eftersom signalen inte tilldelas förrän i slutet av processen

Mer om processer

- Tilldela inte samma signal i olika processer – krav
- Tilldela bara en signal i en process – rekommendation
  - Leder till att ni inte skriver en jättestor process
  - Mindre risk att ni får negativa sidoeffekter
  - Tilldela den helst bara en gång – bättre läsbarhet
- (Så klart OK att beskriva flera d-vippor i samma process)

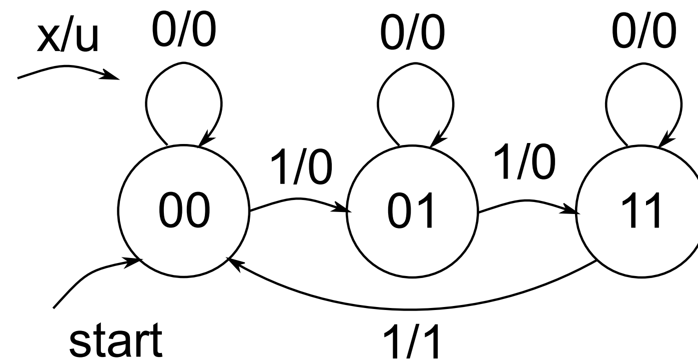
Använd bara kombinatoriska processer om du vet att du behöver det!

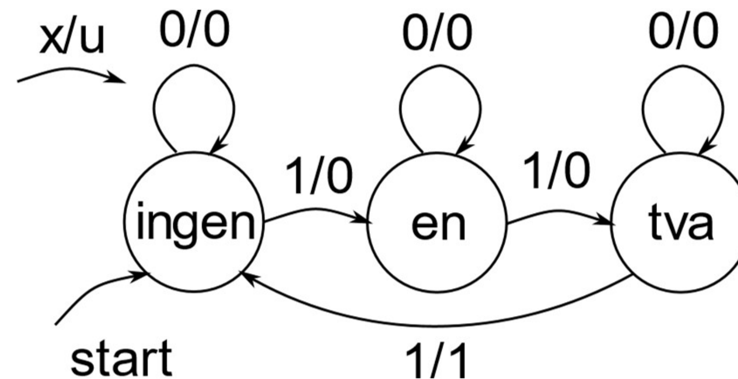
- Ja, det blir mer likt “vanlig” programmering, men vi beskriver ju faktiskt hårdvara
- Nästan varje år så ångrar jag nästan att jag tar upp möjligheten (men annars så googlar ni säkert…)

Egna typer

- Från förra föreläsningen:
  - Det finns i princip inga inbyggda typer i VHDL utan det mesta är definierat
- Finns flera anledningar att definiera egna typer, men något som är användbart nu är för att slippa göra tillståndskodning

Etträknaren med egna typer

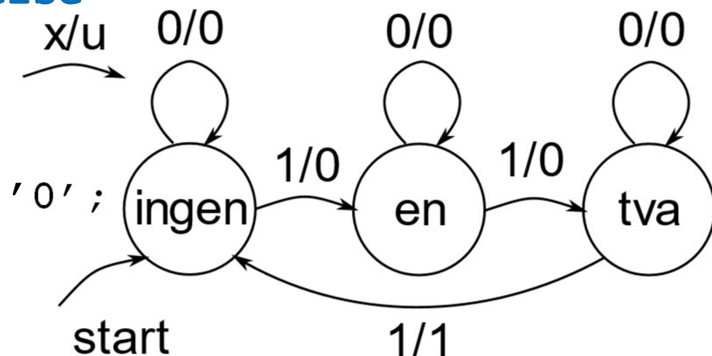




Etträknaren med egna typer

```
architecture ...
type tillstand is (ingen, en, tva);
signal q, qplus : tillstand;
begin
-- q+ = f(q,x): tillståndsberäkning
qplus <= ingen when q = ingen and x = '0' else
         en when q = ingen and x = '1' else
         en when q = en and x = '0' else
         tva when q = en and x = '1' else
         tva when q = tva and x = '0' else
         ingen;

-- u = g(q,x): utsignal
u <= '1' when q = tva and x = '1' else '0';
```



Etträknaren med egna typer

- I simulatorn så står tillståndsnamnet i vågformen
- Vid syntes så översätter verktyget tillståndsnamnet till en binärkodning som går att styra
    - Binärt
    - Gray
    - Slumpat
- (Det har i efterhand införts i standarden att egna typer ska kodas binärt i ordning, men många syntesverktyg har stöd för olika kodningar.)

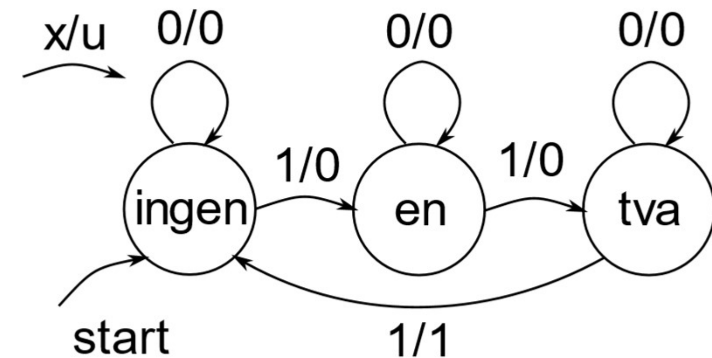
Egna typer

- Kan vara speciellt smidigt i kombination med **case-when**

```
case state is
  when state1 => uppdatera tillstånd;
                 sätt utsignal;
  when state2 => uppdatera tillstånd;
                 sätt utsignal;
 ...
  when stateN => uppdatera tillstånd;
                 sätt utsignal;
end case;
```

Etträknaren med egna typer

```
architecture …
type tillstand is (ingen, en, tva);
signal q, qplus : tillstand;
begin
-- q+ = f(q,x): tillståndsberäkning
process(q, x) begin
case q is
when ingen => u <= '0;
              if x = '0' then
                 qplus <= ingen;
              else
                 qplus <= en;
              end if;
when en    => u <= '0';
              if x = '0' then
                 qplus <= en;
              else
                 qplus <= tv;
              end if;
when tva   => …
```



Använda byggblock

- Ni förväntas dela upp konstruktionen i olika block och beskriva de olika blocken med väl avgränsade VHDL-satser
- Det kommer så småningom finnas anledning att koppla ihop flera olika block
- Det är inget ni behöver göra under labbarna, men ni får

Använda byggblock

- Två steg:
  - Deklararera en `component` mellan `architecture` och `begin`
  - Instantiera och koppla in byggblocket
- 4-bitsadderare

```
entity adder is
port(a, b: in UNSIGNED(3 downto 0);
     s: out UNSIGNED(4 downto 0));
end adder;
```

Använda byggblock

- Använda 4-bitsadderaren

```
architecture ...

component adder
port(a, b: in UNSIGNED(3 downto 0);
     s: out UNSIGNED(4 downto 0));
end component;

signal i1, i2: UNSIGNED(3 downto 0);
signal out1: UNSIGNED(4 downto 0);

begin


A1: adder
port map (a => i1, b => i2, s => out1);
```

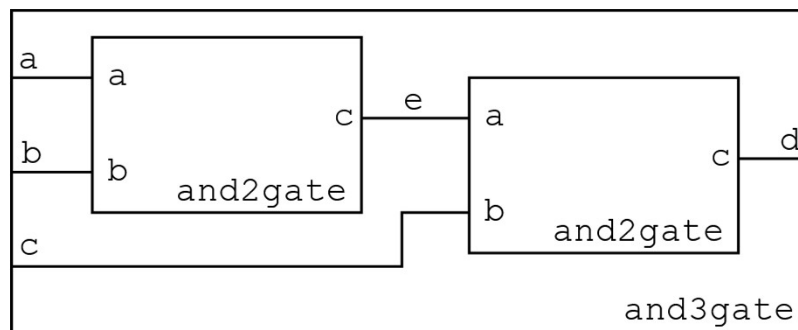
Använda byggblock – komplett exempel

Fil 1: and2gate.vhd

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

entity and2gate is
  port(a, b: in std_logic;
       c: out std_logic);
end entity;

architecture basic of and2gate is
begin  -- architecture basic
  c <= a and b;
end architecture basic;
```



Fil 2: and3gate.vhd

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

entity and3gate is
  port(a, b, c: in std_logic;
       d: out std_logic);
end entity;

architecture basic of and3gate is
  component and2gate is
    port(a, b: in std_logic;
         c: out std_logic);
  end component;
  signal e: std_logic;
begin  -- architecture basic
  and2gate1: and2gate
    port map (a => a, b => b, c => e);
  and2gate2: and2gate
    port map (a => e, b => c, c => d);
end architecture basic;
```

Använda byggblock

- I föregående exempel är blocken på alldeles för detaljerad nivå
- Försök hitta lagom avvägning
- Försök hitta byggblock som går att återanvända
- Går självklart att blanda komponenter, parallella och sekventiella satser i samma arkitektur
  - Allt exekveras parallellt gentemot varandra

Använda byggblock

- Gäller att kompilera/syntetisera filerna i rätt ordning
- "Nerifrån och upp", dvs i föregående exempel
  - Först and2gate.vhd
  - Sedan and3gate.vhd
- and3gate.vhd kommer att leta efter en redan kompilerad entity som heter and2gate (oavsett vad filen heter)
- Verktygen är mer eller mindre känsliga för och hjälpsamma med detta

Generella byggblock

- 4-bitsadderare

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

entity adder is
port(a, b: in UNSIGNED(3 downto 0);
     s: out UNSIGNED(4 downto 0));
end adder;

architecture simple of adder is
begin
   s <= resize(a, 5) + resize(b, 5);
end simple;
```

Generella byggblock

- 5-bitsadderare

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

entity adder is
port(a, b: in UNSIGNED(4 downto 0);
     s: out UNSIGNED(5 downto 0));
end adder;

architecture simple of adder is
begin
  s <= resize(a, 6) + resize(b, 6);
end simple;
```

Genererella byggblock

- N-bitsadderare?
- Vore ju smidigt om vi slapp skapa en fil för varje längd...
- Vi kan använda **`generic`**!

```
entity adder is
generic(N: integer := 4);
port(a, b: in UNSIGNED(N-1 downto 0);
        s: out UNSIGNED(N downto 0));
end adder;
```

Generella byggblock

- N-bitsadderare

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

entity adder is
generic(N: integer := 4);
port(a, b: in UNSIGNED(N-1 downto 0);
     s: out UNSIGNED(N downto 0));
end adder;

architecture simple of adder is
begin
  s <= resize(a, N+1) + resize(b, N+1);
end simple;
```

Använda generella byggblock

- N-bitsadderare

```
component adder
  generic(N: integer);
  port(a, b: in UNSIGNED(N-1 downto 0);
       s: out UNSIGNED(N downto 0));
end component;


. . .


A1: adder
  generic map (N => 8)
  port map    (a => i1, b => i2, s => out1);
```

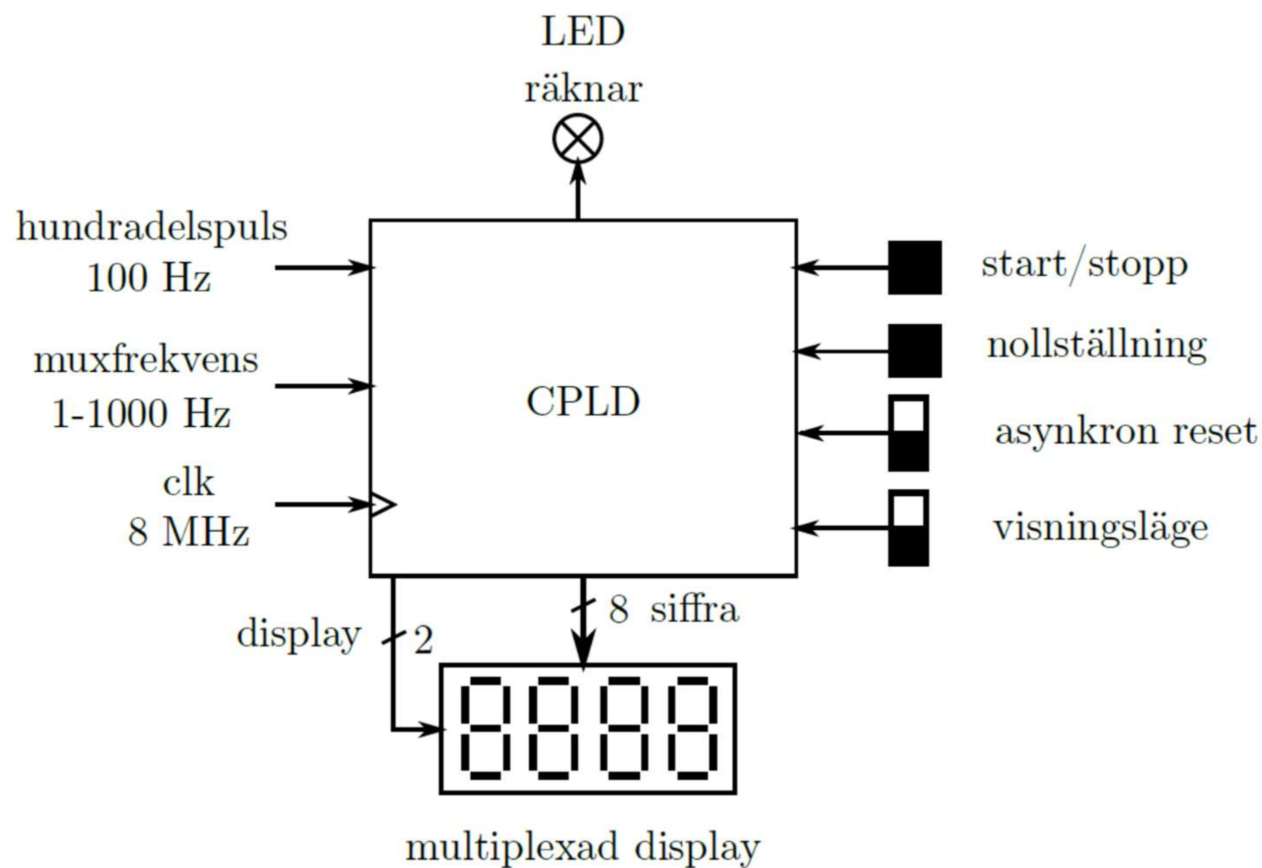
Rekommendation

- Använd endast `std_logic` och `std_logic_vector`
- Vill ni räkna inkludera `numeric_std` –biblioteket
- Skriv logik och vippor separat så långt det går
- Skriv kombinatorik med tilldelningar (utan processer)
- Använd bara kombinatoriska processer om ni är övertygade om att det ger stora fördelar (och är beredda att förstå konsekvenserna fullt ut)
- `component` och `generic` är användbara koncept, men långt ifrån nödvändigt i labbarna
- **Kom ihåg att ni beskriver hårdvara!**

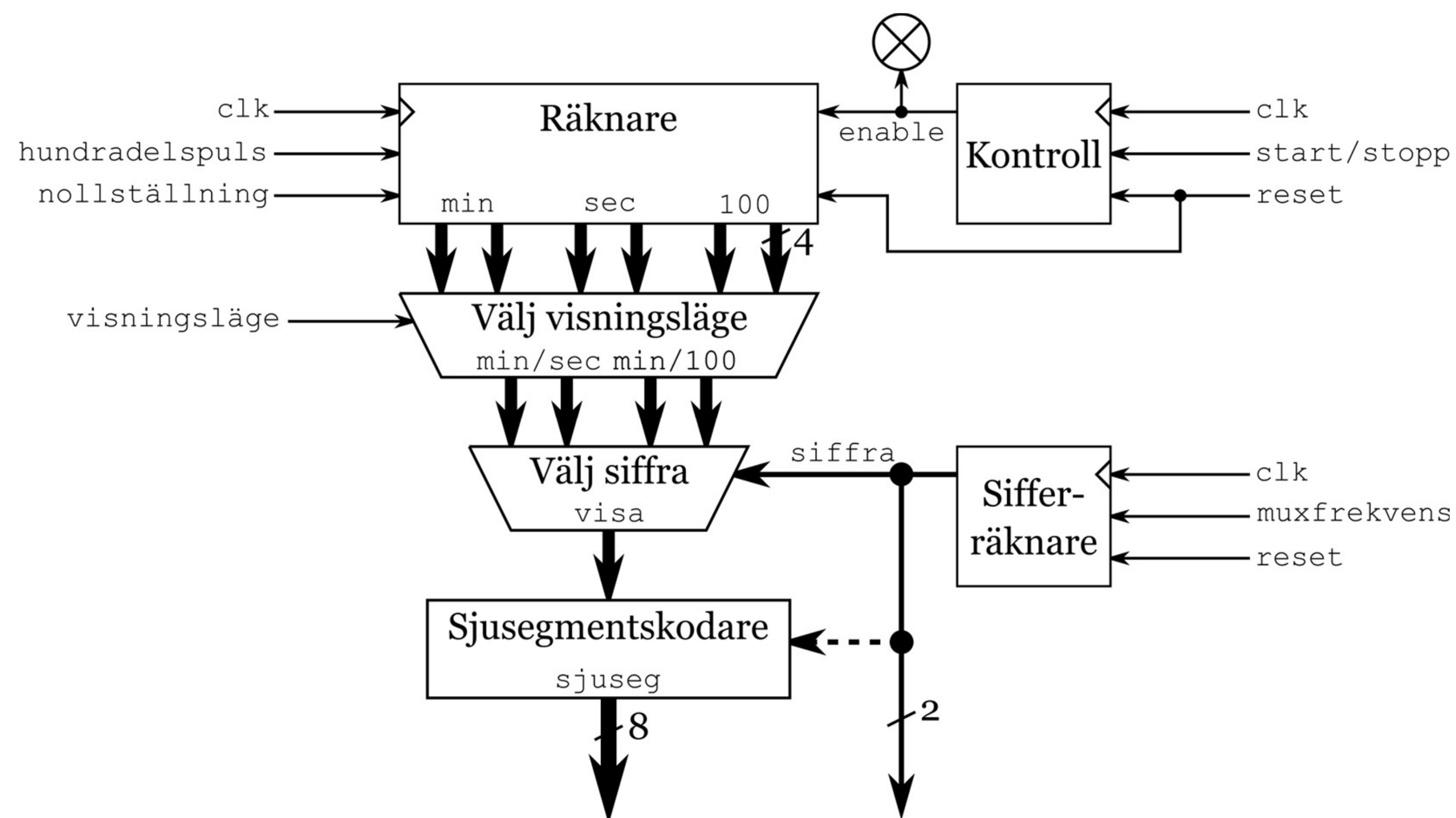
Lab 4

- Fortsättningslektion på tisdag (se den som labbförberedelse med lätt åtkomligt stöd om ni är klara med det lektionsmaterialet)
- Uppgift 4.1 – inklusive simuleringsföreberedelseuppgift
- Minst en av uppgift 4.2 och 4.3 – inklusive blockschema
- Finns testbänkar att simulera med för att kontrollera korrekt funktion

Uppgift 4.1 : Timer



Uppgift 4.1

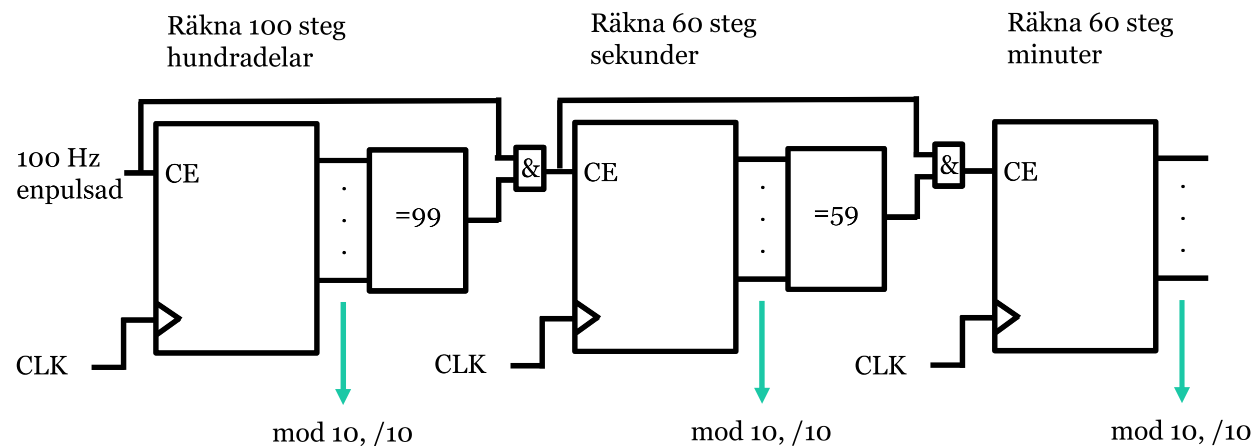


Uppgift 4.1

- Alternativ 1 för räknaren
  - Räkna hundradelar (ser enkelt ut, men ...)
    - 19-bitars räknare (360000 steg)
    - mod 10, /10, mod 10, /10, mod 10, /10, mod 6, /6, mod 10, /10
    - T ex 247892 hundradelar
      247892 mod 10 = 2 (entals hundradelar)
      247892 / 10 => 24789 mod 10 = 9 (9 tiotals hundradelar)
      24789 / 10 => 2478 mod 10 = 8 (entals sekunder)
      2478 / 10 => 247 mod 6 = 1 (tiotals sekunder)
      247 / 6 => 41 mod 10 = 1 (entals minuter)
      41 / 10 = 4 (tiotals minuter)

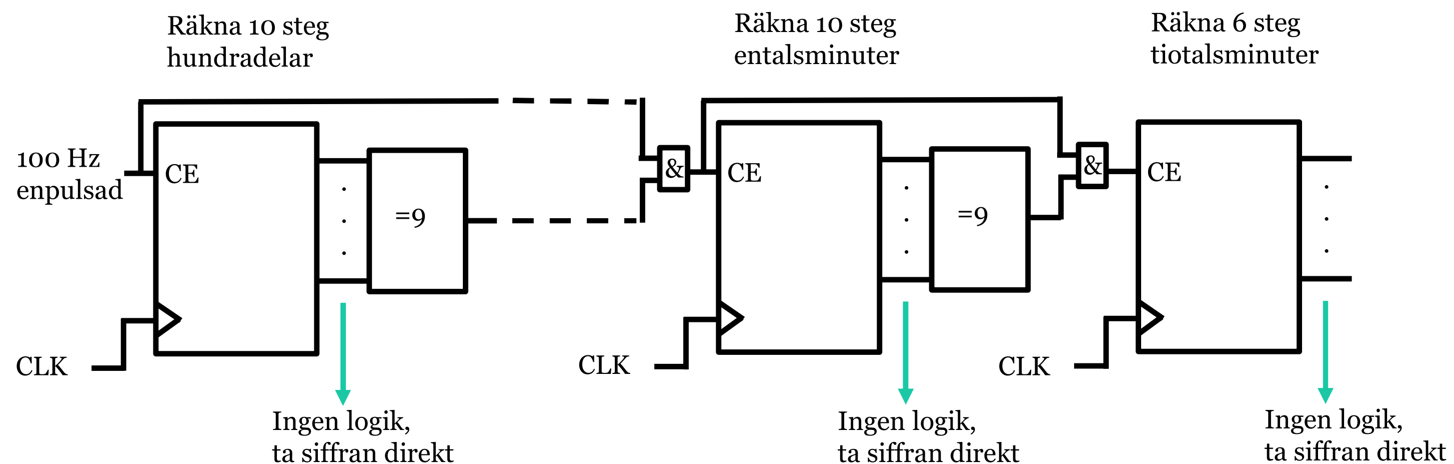
Uppgift 4.1

- Alternativ 2 för räknaren
  - Räkna hundradelar, sekunder, minuter (bättre? men ...)
    - 2 st 6-bitars räknare (60 steg), en 7-bitar räknare (100 steg), totalt 19 bitar
    - 3 st mod 10, /10



Uppgift 4.1

- Alternativ 3 för räknaren, forts
  - Räkna hundradelar, tiondelar, sekunder, tiotals sekunder, minuter, tiotals minuter (BCD-kodning)
    - 4 st 4-bitars räknare (10 steg), 2 st 3-bitars räknare (6 steg), totalt 22 bitar
    - Ingen övrig logik (för avkodning av siffror)



OBS! Entalsminuter ska räkna upp när tiotalssekunder=5 OCH entalssekunder=9 samt att sekunder räknas upp (CE=1). Entalssekunder ska räkna upp när tiondelar=9 samt att tiondelar räknas upp. Dvs, olika villkor för olika räknare.

Uppgift 4.1

- Division och modulo
  - Sök på numeric_std för att hitta numeric_std.vhd

```
 -- Id: A.23
  function "/" ( L: UNSIGNED; R: NATURAL) return UNSIGNED;
     -- Result subtype: UNSIGNED (L'LENGTH-1 downto 0)
     -- Result: Divides an UNSIGNED vector, L, by a non-negative INTEGER, R.
     --         If NO_OF_BITS(R) > L'LENGTH, then R is truncated to L'LENGTH.

...

 -- Id: A.35
  function "mod" ( L: UNSIGNED; R: NATURAL) return UNSIGNED;
     -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)
     -- Result: Computes "L mod R" where L is an UNSIGNED vector and R
     --         is a non-negative INTEGER.
     --         If NO_OF_BITS(R) > L'LENGTH, then R is truncated to L'LENGTH.
```

- **Kan kräva mycket logik**

“Två klockor”

- Det finns till synes två “klockor” till vissa block
  - **`clk`** och **`hundradelspuls`** till räknaren
  - **`clk`** och **`muxfrekvens`** till sifferräknaren

```
if rising_edge(clk) and rising_edge(hundradelspuls)
```

Fungerar inte!

- Finns ingen komponent som fungerar så (vippor har **en** klocka)
- Sannolikheten att flankerna kommer exakt samtidigt är minimal

“Två klockor”

- Egentligen bara en klocka: **clk**
  - Vi vill ha ett synkront system
- Rätt sätt

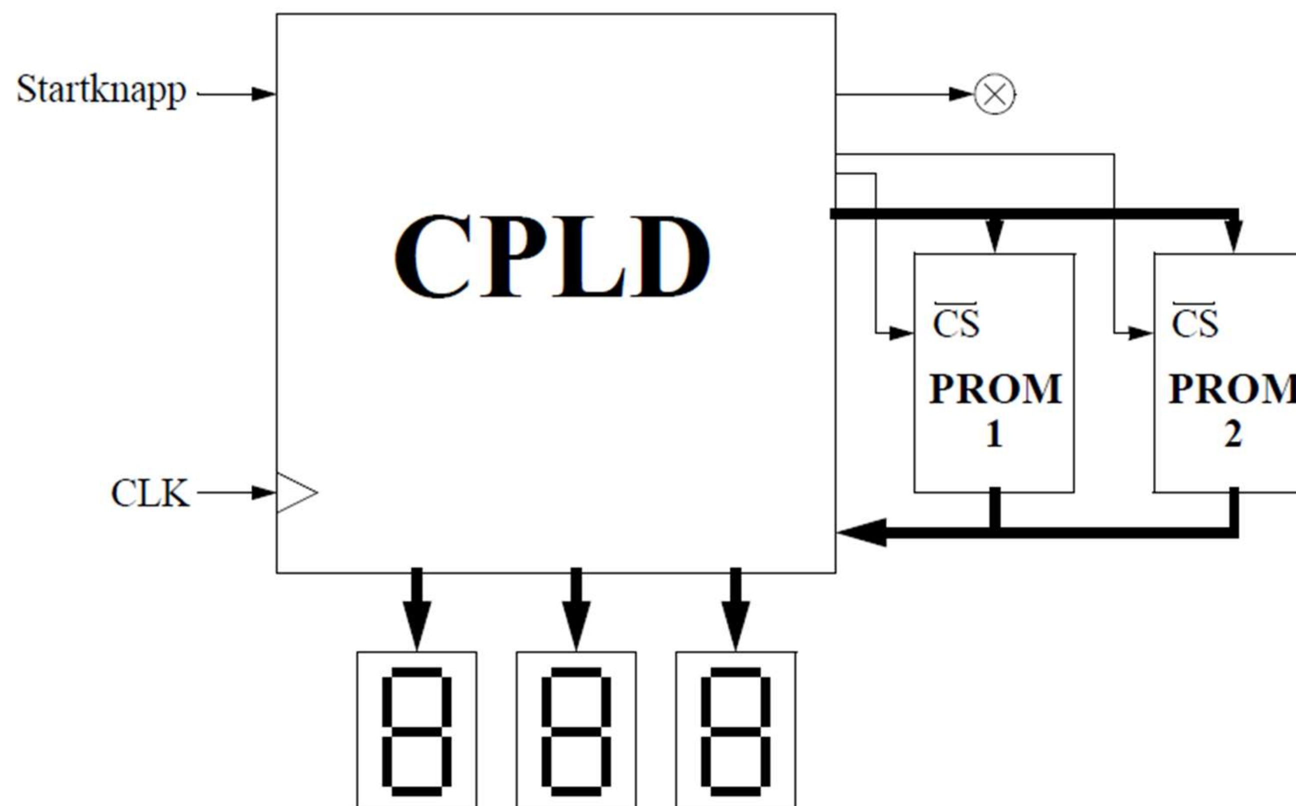
```
-- synkronisera hundradelspuls
process(clk) begin
    if rising_edge(clk) then
        sync_hundra1 <= hundradelspuls;
        sync_hundra2 <= sync_hundra1;
    end if;
end process;

-- enpulsad hundradelspuls, kopplas till CE
enpuls_hundra <= sync_hundra1 and not sync_hundra2;
```

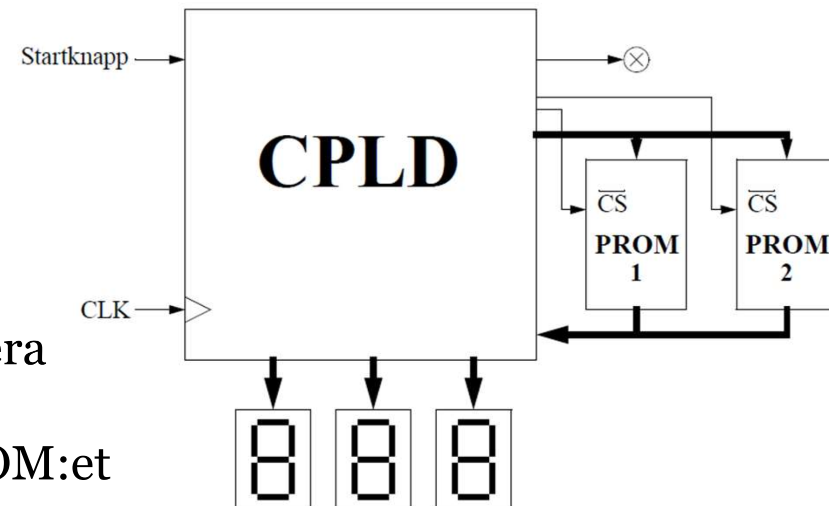
Simulera i ModelSim

- Ni avgör själva om räknarna i 4.1 har load, clear, osv
  - Lättare att bygga vidare på till senare deluppgifter
  - Krångligare att få till 4.1 och syftet med 4.1 uppnås även utan dessa

Uppgift 4.2 : Räkna 1:or i två PROM

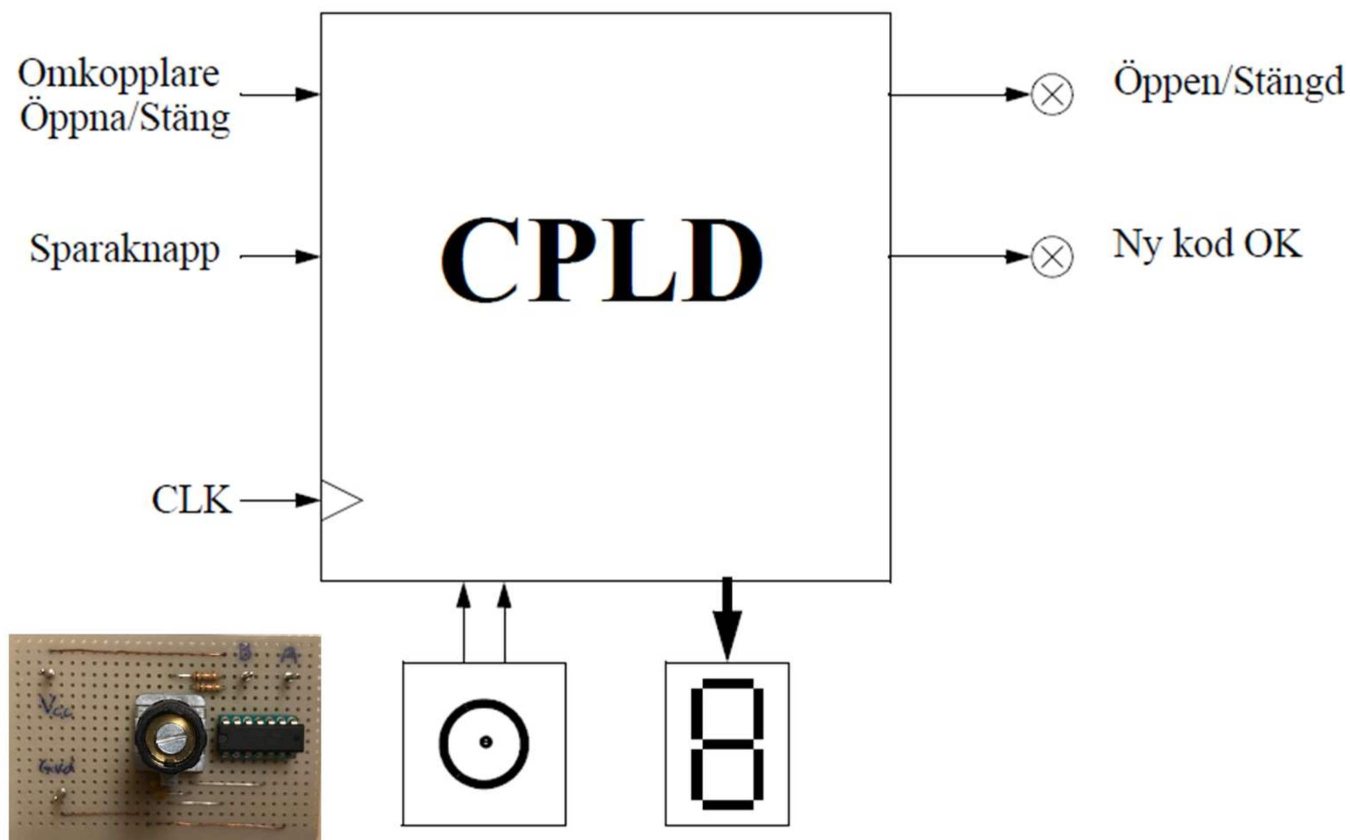


Uppgift 4.2 : Räkna 1:or i två PROM



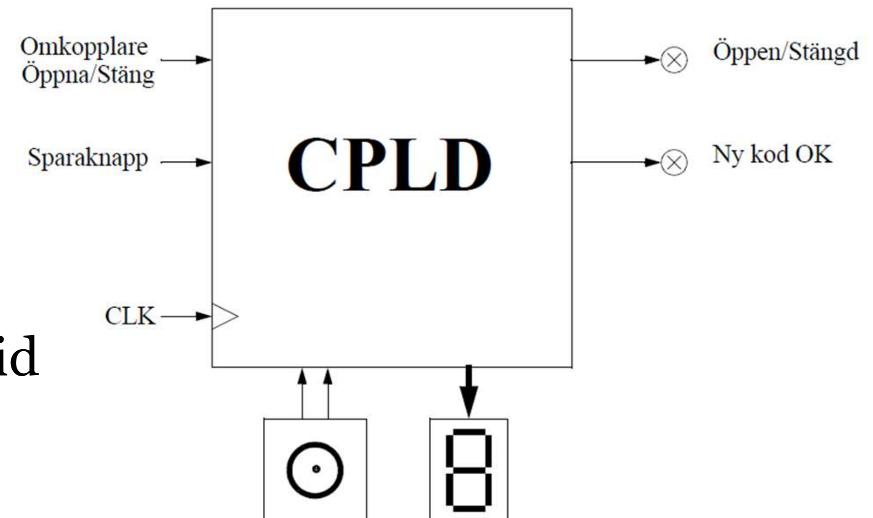
- Flera delar liknar uppg 4.1
  -Muxad display
  -Enpulsad uppräkning
- Två PROM kan betraktas som ett enda (dubbelt så stort)
  -Använd 5 bits räknare för att adressera
  -Invertera mest signifikant bit till ena PROM:et, oinverterad till andra PROM:et
- Den kan vara 0 till 4 1:or per address i PROM:et
  -Skifta/muxa 4-bits ord och räkna 1:or
  (dvs använd 7-bits räknare, två minst sign. bitarna muxar ordet)
  -Översätt 4-bits ord till antal ettor, och addera till resultatet
  (dvs bygg en "BCD-adderare", komplexare lösning som kanske inte får plats?)

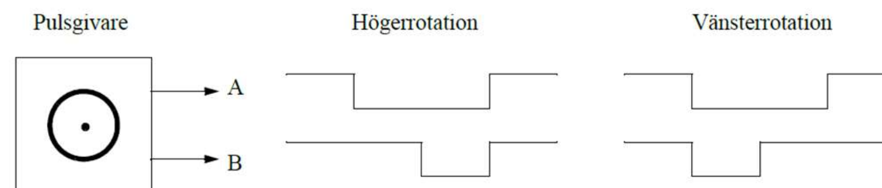
Uppgift 4.3 : Kassaskåpslås



Uppgift 4.3 : Kassaskåpslås

- Två pulståg måste avkodas
  -Ordningen på pulserna avgör
   rotation åt höger eller vänster
- Tvåsiffrig kod
- Alt1: Vrid fram siffra 1, spara, vrid fram siffra 2, spara, öppna med omkopplare
- Alt2: Vrid höger till siffra 1, vrid vänster till siffra 2, vrid höger till 0, öppna med omkopplare





Digitalteknik

Oscar Gustafsson
Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET