

# Programmerbara kretsar och VHDL 2

Föreläsning 10

Digitalteknik, TSEA22

Oscar Gustafsson

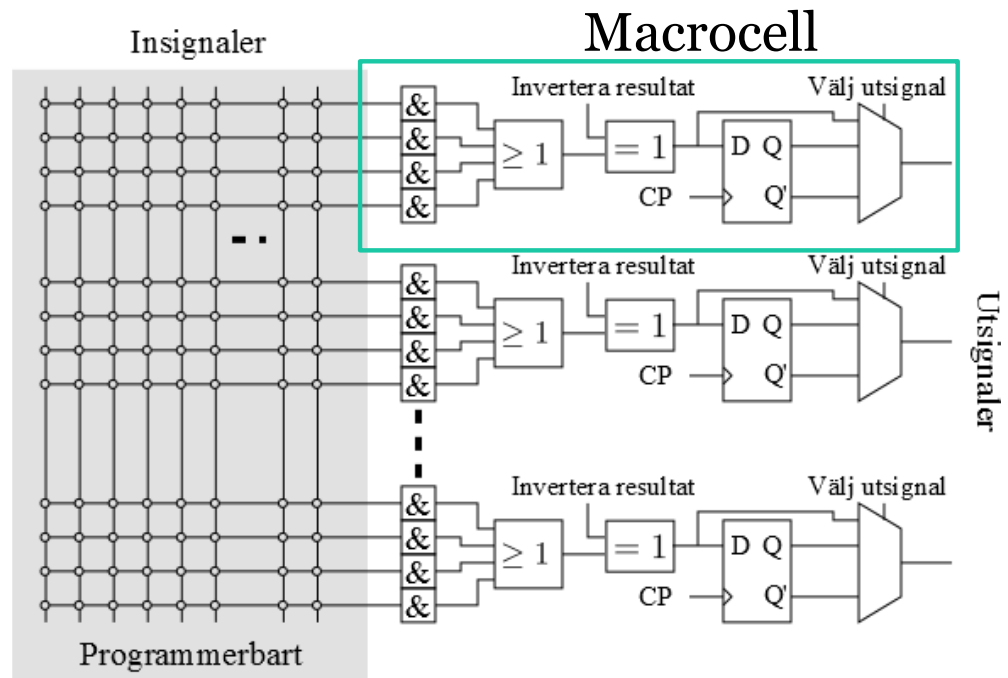
Institutionen för systemteknik

# Dagens föreläsning

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

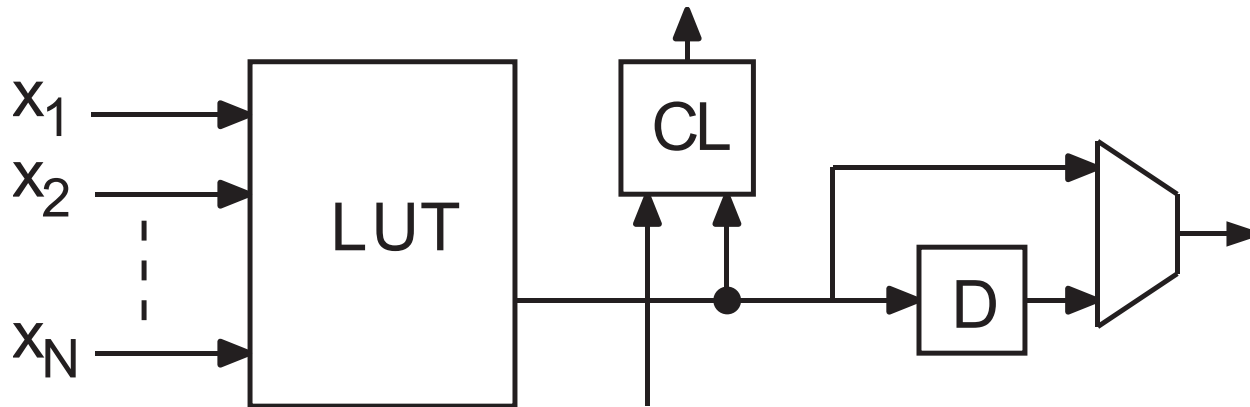
# Programmerbara kretsar

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



# FPGA

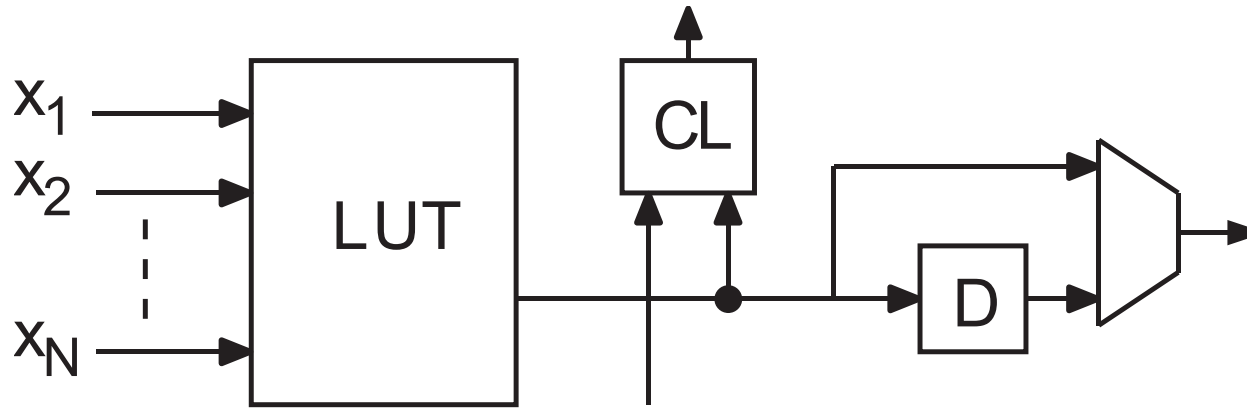
- PROM/tabell/minne går att använda till alla kombinatoriska kretsar
- FPGA – programmerbar krets med en massa tabeller



# 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



```
-- räknare
ctr16: process (clk, reset)
begin
    if reset = '1' then          -- asynkron reset
        q <= "0000";
    elsif rising_edge (clk) then
        q <= qplus;
    end if;
end process ctr16;
```

# 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;
```

# 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;
end process;
```

- Utsignalen tilldelas inte alltid

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

# Vanliga fel i kombinatoriska processer

- Simulatore 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
- Några konsekvenser av detta är:
  - En signal kan tilldelas flera gånger i processen
  - Ett nytt tilldelat värde kan inte användas senare i processen
  - 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;
        if c > 2:
            a <= c;
        end if;
    end if;
end process;
```

fungerar utan problem

# Jämför

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

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

Två grindar driver samma signal!

# 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;
```

Sämre kod!

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

Bättre kod!

Det är att föredra att en variabel bara tilldelas en gång



# Mer om processer

- Själva tilldelningen sker i slutet av processen

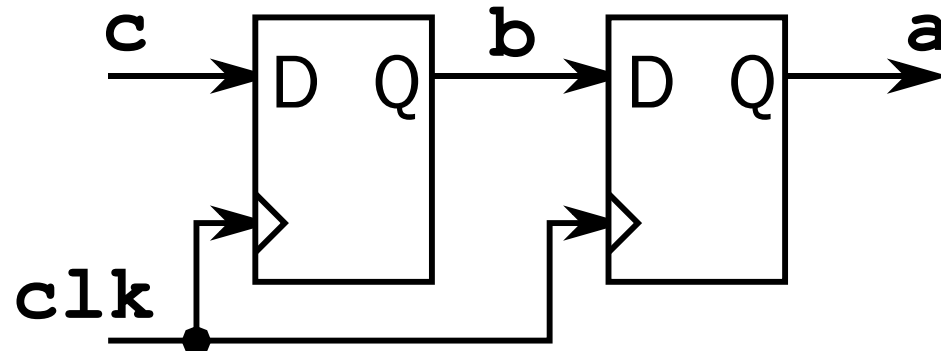
```
process (clk)
begin
    if rising_edge (clk) then
        b <= c;
        a <= b;
    end if;
end process;
```

- 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;
    a <= b;
  end if;
end process;
```

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

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

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

# Mer om processer

- Tilldelning sker bara i slutet oavsett om processen är klockad eller inte

```
process (c, d) - Kör vid ändring av c, d
begin
    b <= c and d;
    a <= b or c;
end if;
end process;
```

- Kommer inte göra det du tror
- **process (b, c, d)** hjälper, men dåligt sätt generellt (körs två gånger)

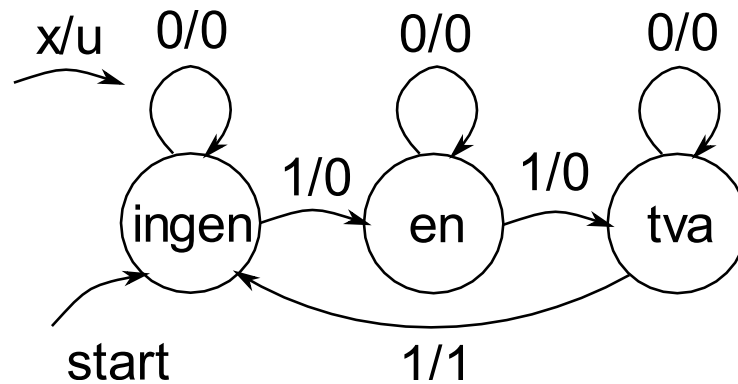
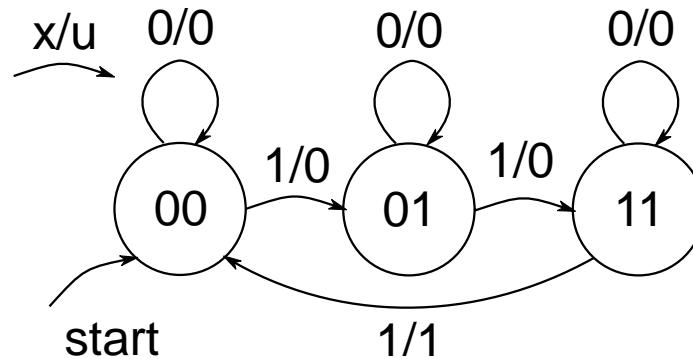
# Använd bara kombinatoriska processer om du vet att du behöver det!

- Ja, det blir mer likt “vanlig” programmering, men vi beskriver ju faktiskt hårdvara
- Förra året ångrade jag nästan att jag tog upp möjligheten

# 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



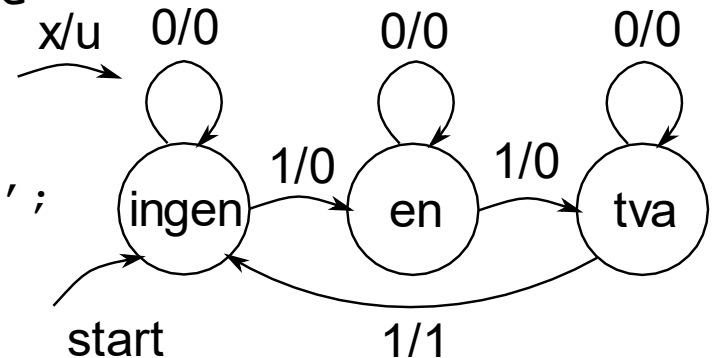
# Ettrräknaren med egna typer

```

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 when q = tva and x='1';

-- 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;
```

# 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

# Använda byggblock

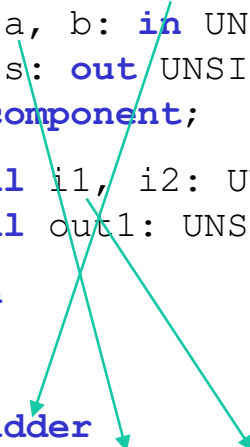
- Två steg:
  - Deklararera en component mellan architecture och begin
  - Instantiera och koppla in byggblocket
- 4-bitsadderare

```
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;

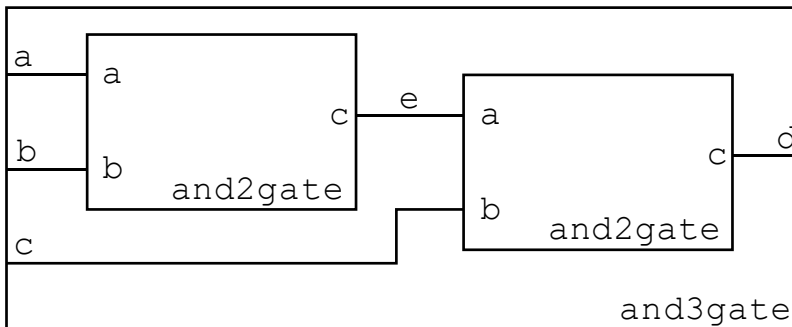
architecture basic of and2gate is
begin -- architecture basic
  c <= a and b;
end architecture basic;
```

## 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;
```

# Generella 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);
```

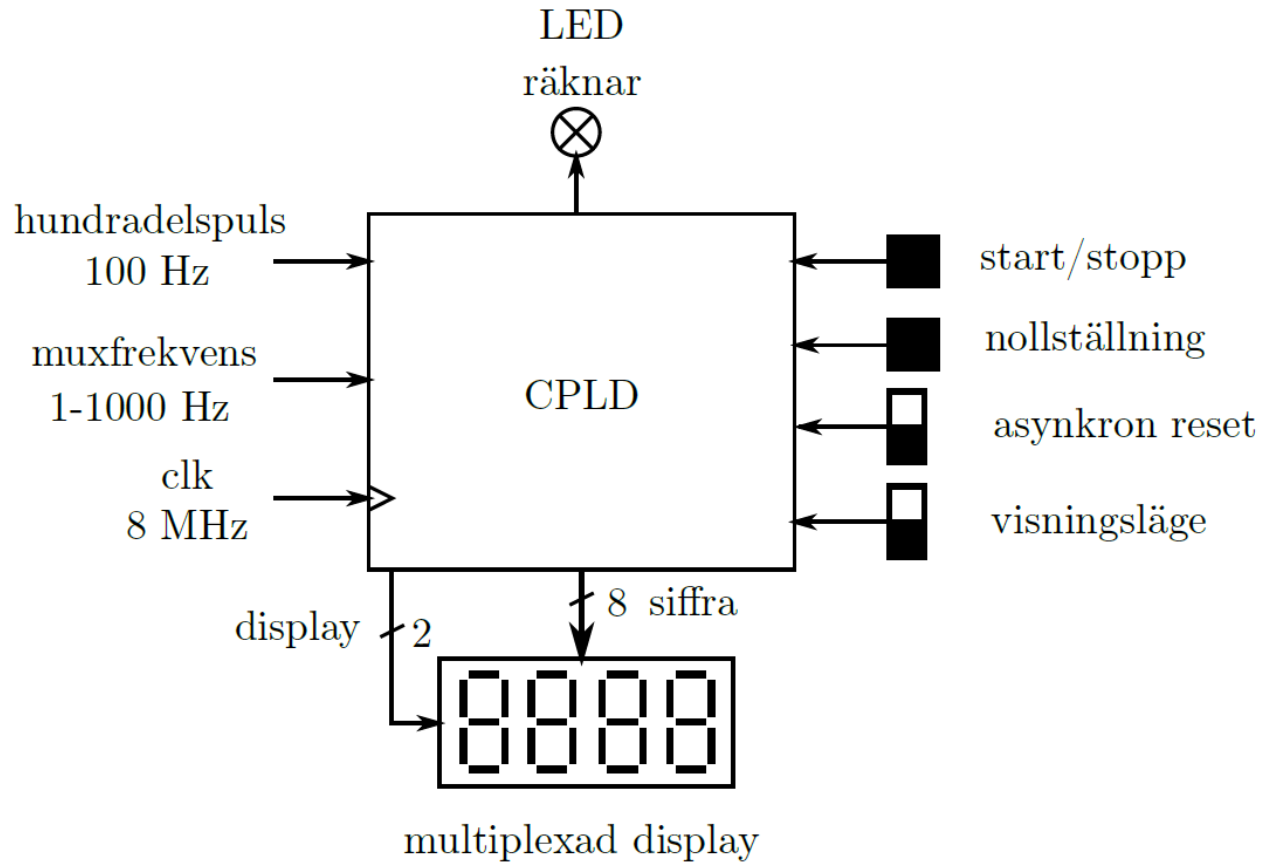
# Rekommendation

- Använd endast `std_logic` och `std_logic_vector`
- Vill ni räkna inkludera `numeric_std` –biblioteket
- Skriv logik och vippor separat så långt det går
- 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

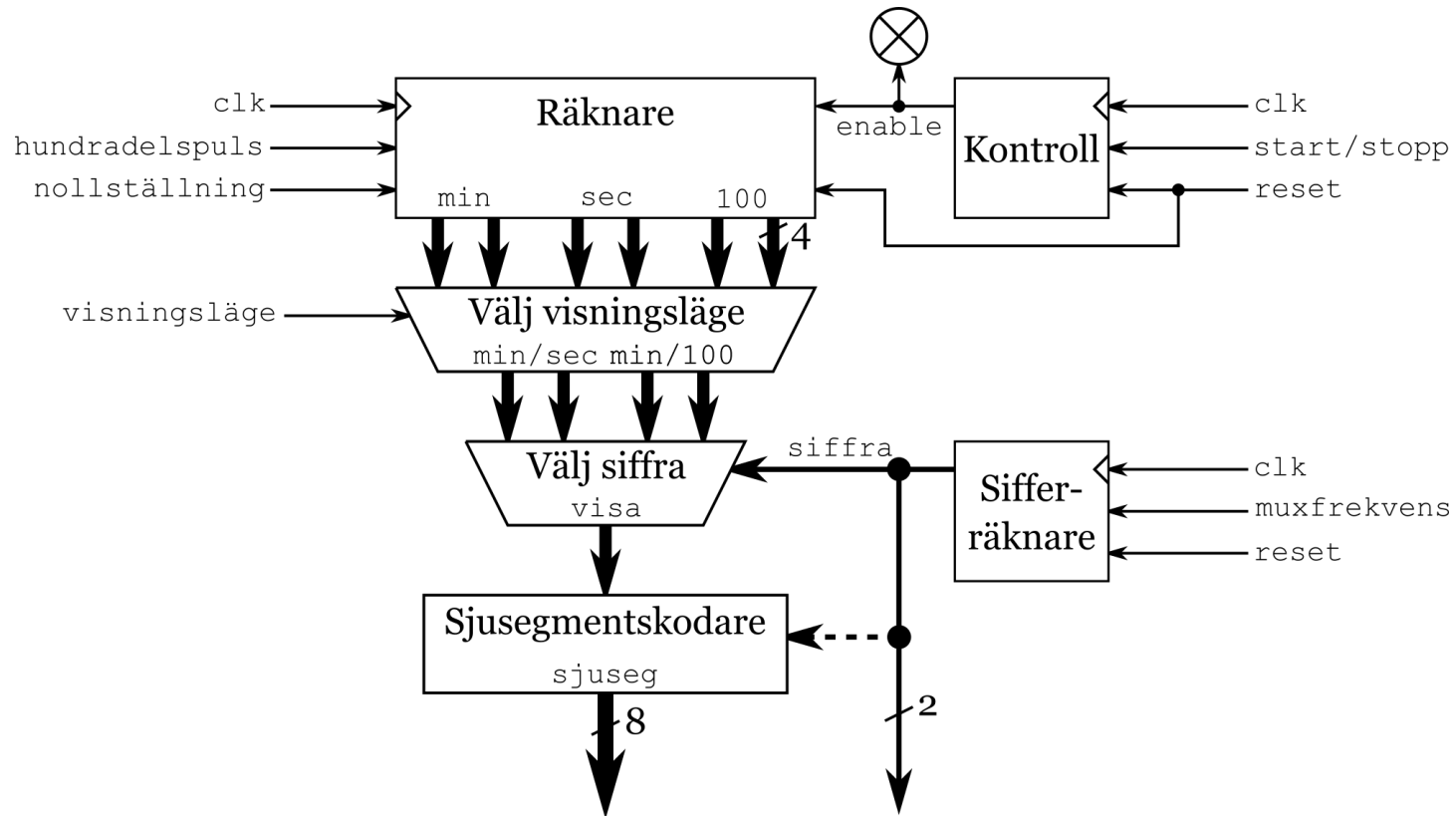
- Obligatorisk lektion i ModelSim (måndag och tisdag)
- Uppgift 4.1 – inklusive simuleringsförebereedelseuppgift
- Minst en av uppgift 4.2 och 4.3
- För uppgift 4.2 finns en testbänk att simulera med för att kontrollera korrekt funktion

# Uppgift 4.1





# Uppgift 4.1



# Uppgift 4.1

- Alternativ för räknaren
  - Räkna hundradelar
    - 19-bitars räknare (360000 steg)
    - mod 10, /10, mod 10, /10, mod 10, /10, mod 6, /6, mod 10, /10
  - Räkna hundradelar, sekunder, minuter
    - 2 st 6-bitars räknare (60 steg), en 7-bitars räknare (100 steg), totalt 19 bitar
    - 3 st mod 10, /10

# Uppgift 4.1

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

# 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
- Sannolikheten att flankerna kommer exakt samtidigt är minimal

# “Två klockor”

- Rätt sätt

```
if rising_edge(clk) then
    -- Uppdatera gamla värdet
    old_hundradelpuls <= hundradelpuls;
    if old_hundradelpuls = '0' and hundradelsplus = '1' then
        -- Gör något fiffigt

    end if;
end if;
```

# Digitalteknik

## Oscar Gustafsson

[www.liu.se](http://www.liu.se)