

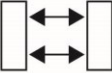
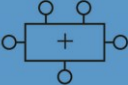

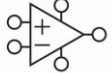




F4 IN2060 2022  
HDL



Yngve Hafting, [yngveha@ifi.uio.no](mailto:yngveha@ifi.uio.no)

Application Software	<code>&gt;"hello world!"</code>
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

# Formål

## • Kort om emnet

- Emnet tar for seg **prinsipper i digital design, som kombinatorisk** og sekvensiell logikk, tilstandsmaskiner og **digitale byggeblokker**, og bygger på dette for å introdusere prosessorarkitekturer, pipelining, cache, og grensesnittet mellom maskinvare og programkode.

## • Hva lærer du?

- Etter å ha tatt IN2060 har du:
  - kunnskaper om hvordan en datamaskin er satt sammen og fungerer, **fra logiske porter til prosessor**
  - kunnskaper om grensesnittet mellom maskinvare og programvare
  - **lært å kunne analysere og konstruere digitale kretser**

## • Delmål

- Kjenne til forskjeller mellom HDL og andre programmeringsspråk
- Kunne modifisere HDL-kode og forstå virkemåten til enkle digitale kretser bygget med HDL
- Kunne utføre enkle simuleringer med simuleringsverktøy for HDL
- Få øvelse i å benytte verktøy for digital design

## • Hvordan?

### – Forelesning:

- Hva mener vi med digitale design?
- Hvorfor HDL?
- Grunnprinsipper VHDL
  - Entitet arkitektur
  - Grunnleggende syntaks
- Litt om kodestiler
- Dataflyt og Strukturell koding
- Testbenker

### – Lærebok:

- Overlapper og utdyper...

### – Ukeoppgaver

- Øvelse i å tolke og skrive HDL

### – Oblig

- Øvelse i å benytte verktøy
- Øvelse i å skrive og forstå HDL
- Øvelse med dataflyt og strukturell koding
- Øvelse i å skrive enkle testbenker

### – Gruppetime(r)

- HDL eksempler, digitale byggeblokker
- Gjennomgang av oppgaver,

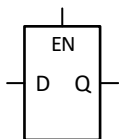
## Fra forrige gang

- Forskjellige «vipper»

latch og flippflopp:

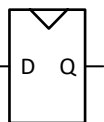
– Latch / Låsevippe:

- Nivå



– Flippflopp (FF)

- Klokkeflanke



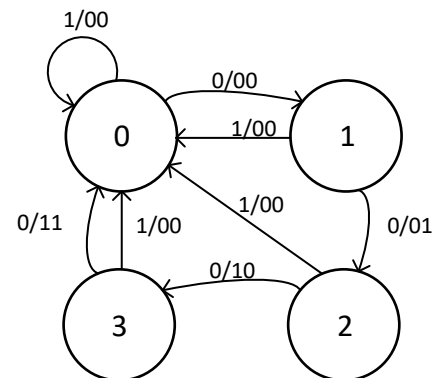
– NB: Noen steder brukes navnene om hverandre = upraktisk

- I kurs(ene) skiller vi dette, alltid.
- I IN2060 skiller vi på symbolene også

– Som regel brukes kun FF'er

- *Latcher er noe vi ikke ønsker*

- Tilstandsmaskiner og tabeller
- Husk
  - Inngang/utgang



## Dagens tema...

- Fra før:
  - Vi kan beskrive...
    - kombinatoriske kretser med ligninger (og skjema)
    - Sekvensielle kretser med skjema og diagram
- Hvordan kan vi lage og simulere digitale kretser?
  - => Hardwarespråk, HDL = Hardware description language

# HDL (VHDL, Verilog)

- Beskriver digitale kretser

  - definerer *virkemåten*

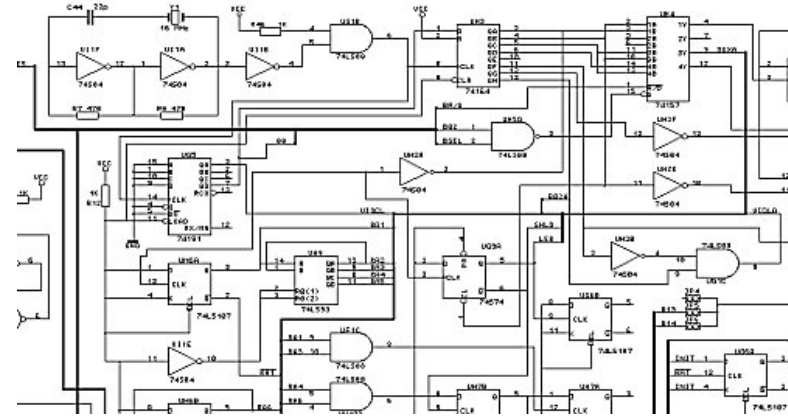
  - $\neq$  Programspråk

    - *programspråk kjøres (linje for linje)*

- Digitale kretser vil alltid være tilstede og "på"

  - hardwarespråk *kan ligne softwarespråk,*

  - men **formålet er fundamentalt forskjellig...**

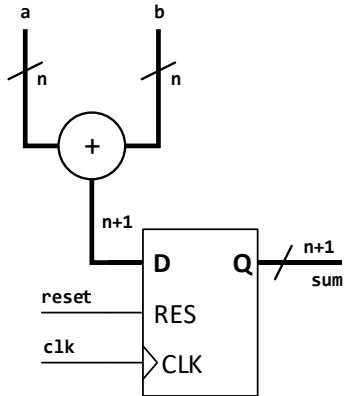


# Hardwarespråk vs Programspråk

```

process(reset, clk)
begin
  if (reset = '1') then
    sum <= '0';
  elsif rising_edge(clk) then
    sum <= a + b;
  end if;
end process;

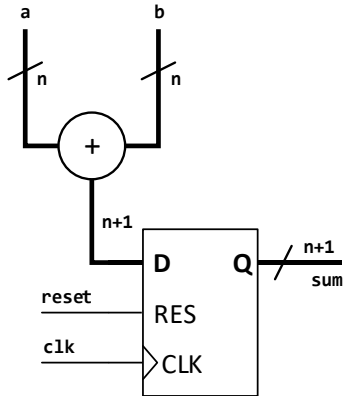
```



HDL «Hardware description language»	Software «Vanlig programspråk»
	<pre> int sum(int a, int b){   return a + b ; } </pre>
	<pre> MOV R5, #0      ;set base adr LDR R7, [R5, #8] ;load reg R7 LDR R8, [R5, #12];load reg R8 ADD R0, R7, R8  ;R0=R7+R8 STR R0, [R5, #16];store R0 </pre>
	<pre> 01001100 10011001 00100100 01001010 11001100 10111001 01100100 11110110 01001100 10011001 00111100 11101010 01001100 10011001 00100100 01011010 ... </pre> <p>(binærkoden tilfeldig sammensatt for illustrasjonsformål)</p>

# Hardwarespråk vs Programspråk

```
process(reset, clk)
begin
  if (reset = '1') then
    sum <= '0';
  elsif rising_edge(clk) then
    sum <= a + b;
  end if;
end process;
```



HDL «Hardware description language»	Software «Vanlig programspråk»
Definerer <b>logisk funksjon</b> til en <b>krets</b>	Definerer <b>instruksjoner</b> og deres rekkefølge for en <b>prosessor</b>
CAD (DAK) verktøy <b>syntetiserer designet</b> slik at det kan realiseres med fysiske porter.	<b>En kompilator oversetter programmet til maskinkode</b> som prosessoren kan kjøre sekvensielt fra minne.
<b>Implementeres i programmerbar logikk (PL) eller applikasjons-spesifikke kretser (ASIC)</b> (PL = FPGA, CPLD, PLD, PAL, PLA, ...) (ASIC = hva-som-helst-slags-chip: prosessorer, mikrokontrollere, osv.)	<b>Lagres i minnet til en datamaskin</b>
<b>Verilog (SystemVerilog)</b> <b>VHDL (VHDL 2008)</b>  (System C m. fl.)	C, C++, C#, Python, Java, assemblere (ARM, MIPS, x86, ...) Fortran, LISP, Simula, Pascal, osv...

```
int sum(int a, int b){
  return a + b ;
}
```

```
MOV R5, #0 ;set base adr
LDR R7, [R5, #8] ;load reg R7
LDR R8, [R5, #12];load reg R8
ADD R0, R7, R8 ;R0=R7+R8
STR R0, [R5, #16];store R0
```

```
01001100 10011001 00100100 01001010
11001100 10111001 01100100 11110110
01001100 10011001 00111100 11101010
01001100 10011001 00100100 01011010
...
(binærkoden tilfeldig sammensatt for illustrasjonsformål)
```

# Designprosess

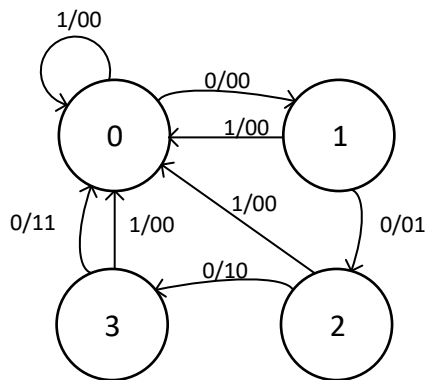
- Noen flere step
- Mer bruk av verktøy
- Simulering kan gjøres for hvert nye steg  
(kompilering, syntese, portplassering og implementering)

Digitale design	Programvare design
Spesifikasjon	Spesifikasjon
Design (koding)	Design (koding)
Kompilering	Kompilering
<b>Simulering</b>	<i>(Simulering/emulering)</i>
<b>Syntese = (port generering)</b>	
<b>Plassering av porter</b>	
<b>Timing analyse. Timing simulering</b>	
<b>Programmering / implementering</b>	<i>Kopiering</i>
Fysisk testing	Testing



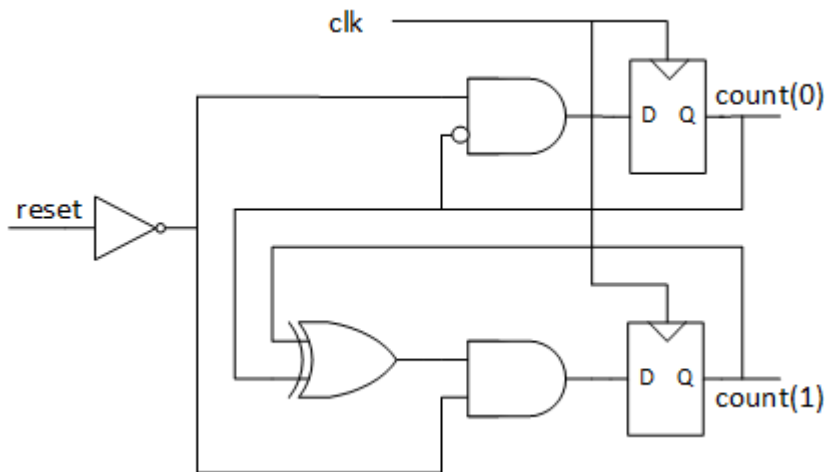
## HDL vs skjemategning:

- Eks: 2 bit teller



- Tilstandsmaskin /  
Tilstandsdiagram

- Én implementasjon, skjema:

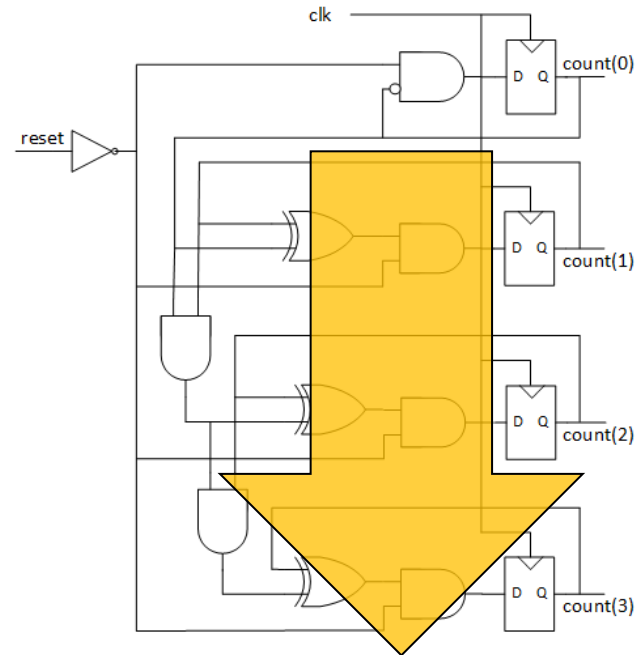


- Hardwarespråk (høynivå-RTL):

```
generic (N : positive := 2);  
---  
count : out std_logic_vector (N-1 downto 0);  
---  
signal next_count : std_logic_vector (N-1 downto 0);  
---  
-- register assignment  
count <= next_count when rising_edge(clk);  
  
-- combinational assignments  
next_count <=  
(others => '0') when reset else  
std_logic_vector( unsigned(count)+1 );
```

## HDL vs skjema...

- Tegn 4 bit teller,  $N = 4$ :
  - Tilstandsdiagram..?
  - Prøv: 2 min...
  - Portskjema..?
  - VHDL kode:  
*Enkelt å skalere opp design...*



```
generic (N : positive := 4);  
---  
count : out std_logic_vector (N-1 downto 0);  
---  
signal next_count : std_logic_vector (N-1 downto 0);  
---  
-- register assignment  
count <= next_count when rising_edge(clk);  
  
-- combinational assignments  
next_count <=  
(others => '0') when reset else  
std_logic_vector( unsigned(count)+1 );
```

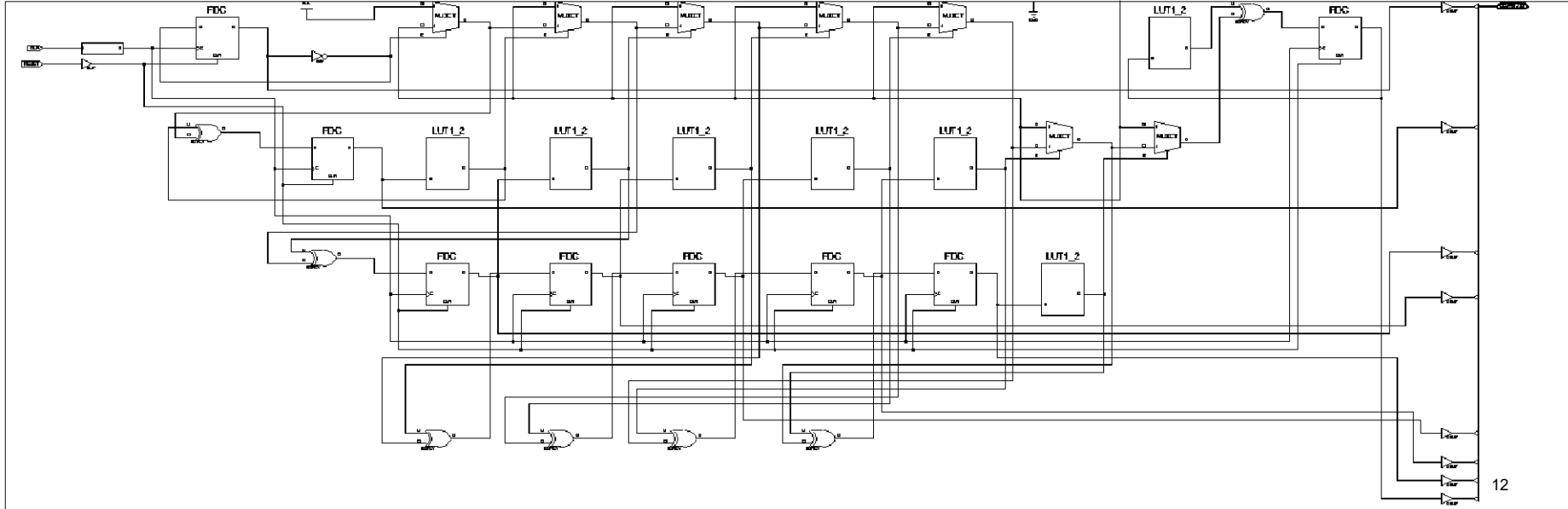
8 bit, ...

```
generic (N : positive := 8);
```

```
generic (N : positive := 8);  
--  
count : out std_logic_vector (N-1 downto 0);  
--  
signal next_count : std_logic_vector (N-1 downto 0);  
--  
-- register assignment  
count <= next_count when rising_edge(clk);  
  
-- combinational assignments  
next_count <=  
  (others => '0') when reset else  
  std_logic_vector( unsigned(count)+1 );
```

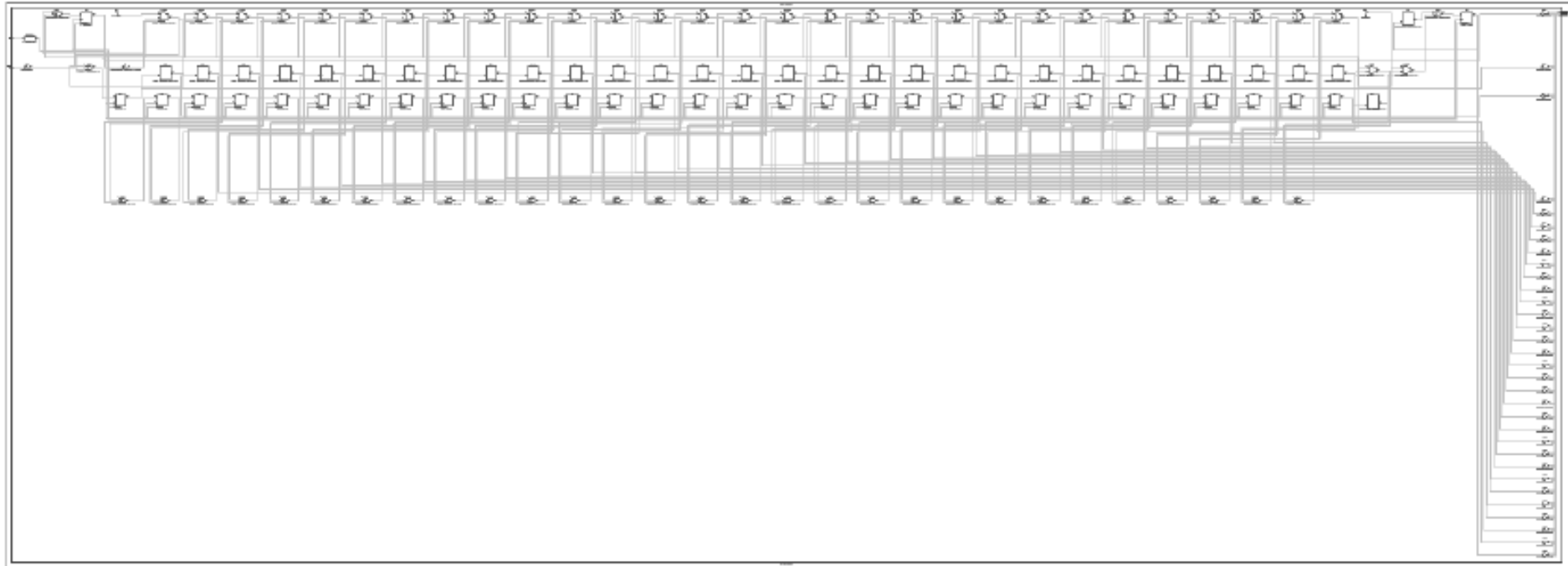
Autogenerert skjema for implementasjon fra synteseverktøy

- Bruker FF'er, LUTer og MUX'er i tillegg til logiske porter og invertere (Skjemaet er kun med for å illustrere kompleksiteten)



32 bit...

`generic (N : positive := 32); --...`



## I dette kurset...

- Enkel bruk av VHDL til simulering
  - *Koding av kombinatoriske kretser*
    - *Lavnivå : Dataflyt, struktur*
  - *Simulering med testbenk*
- VHDL brukes til å forklare datamaskinarkitektur
  - (Vi bruker ikke systemVerilog)

## Senere kurs (IN3160)...

- *realisere fysiske kretser*  
(programmerbar logikk: FPGA )
- Mer om VHDL
  - *Høynivå: RTL-kode*
  - *Mer funksjonalitet*
  - ...
- Mer avanserte design
  - *Kode sekvensielle kretser*
- Mer om skjematikk
- osv...

# HVA GJØR (V)HDL?

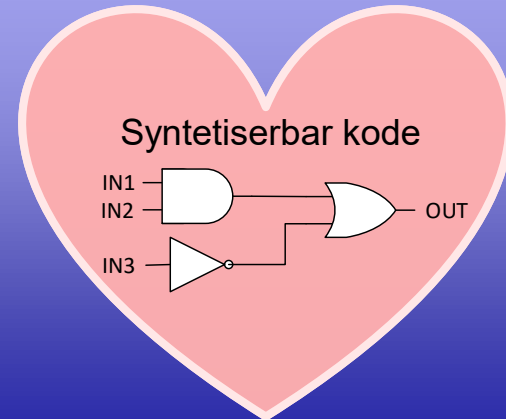
VHDL = VHSIC HDL:

- «Very High Speed Integrated Circuit Hardware Description Language»
- **Formålet er å lage (syntetisere) digitale kretser**
- **Syntetiserbar kode virker i parallell!**
  - Syntetiserbar = kan lages i hardware
  - **Danner kretser, *ikke et program.***
    - Rekkefølge *kan* danne grunnlag for *prioritet*
- Simuleringskode (*testbenker*) kan gjøre mer enn syntetiserbar kode
  - Sette og teste stimuli
  - Rapportere til fil og skjerm
  - kan beskrive hendelser i rekkefølge =>
    - er delvis sekvensielle program...
    - *kan være forvirrende..!*

## VHDL

Kode for å lage testdata og kontrollere testresultater (Testbenker)

Kode for å lage flere instanser eller varianter av enheter



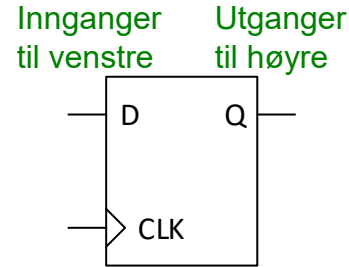
# Krasjkurs i VHDL...:

- Oppbygging
  - Entitet og arkitektur *"Skall" og "innmat"*
- Kodestiler *Ulike måter å beskrive "innmat" og "oppførsel"*
  - dataflow
  - structural
  - RTL
  - behavioral
- Syntaks og konkret koding av design *Hvordan lage beskrivelser som fungerer*
  - Datatyper og verdier
  - Operatorer
  - Tilordning
  - Duplisering av elementer
- Simulering *Testing av oppførsel til kretser*
  - Testbenk *Beskriver forløpet til en test - for en eller flere kretser*

# VHDL oppbygging

## Entitet og arkitektur

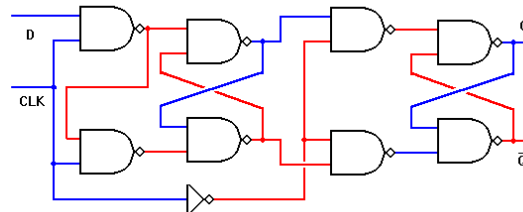
- Entiteten definerer input og output
- Arkitekturen beskriver hva designet gjør.
  - En entitet kan benytte forskjellige arkitekturer (kun én av gangen)
  - Arkitekturer kan defineres med forskjellige stiler (neste slide)
    - “RTL” og “Dataflow” er navn på kodenstil.
      - (mer følger...)



```
library IEEE  
use IEEE.STD_LOGIC.all
```

```
entity D_FLIPFLOP is  
  port(  
    clk : in std_logic;  
    D   : in std_logic;  
    Q   : out std_logic  
  );  
end entity D_FLIPFLOP;
```

```
architecture RTL of D_FLIPFLOP is  
begin  
  process (clk)  
  begin  
    if rising_edge(clk) then  
      Q <= D;  
    end if;  
  end process;  
end architecture RTL;
```



```
architecture data_flow of D_FLIPFLOP is  
  signal e,f,g,h,i,j,k,l : std_logic;  
begin  
  e <= NOT (D and clk);  
  f <= NOT (e and clk);  
  g <= NOT (e and h);  
  h <= NOT (f and g);  
  i <= NOT (g and not clk);  
  j <= NOT (h and not clk);  
  k <= NOT (l and i);  
  l <= NOT (k and j);  
  Q <= k;  
end architecture data_flow;
```

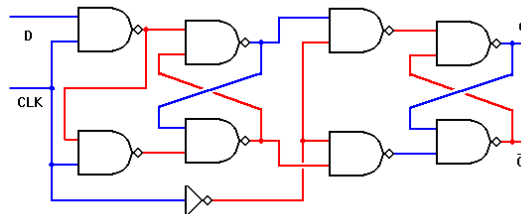


## **4 Kodetyper / stiler:**

- Dataflyt “data\_flow”
- Strukturell “structural”
- RTL “Register transfer level”
- Behavioral

## Dataflyt

- Beskriver (kombinatoriske-) kretser i detalj
  - portnivå «gate level»
    - AND, OR, NAND, NOR, XOR, NOT og koblinger mellom disse og signaler
  - *Ikke bruk av registre*
- Bruksområde:
  - Manuell optimalisering
  - små/ enkle kombinatoriske funksjoner



```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

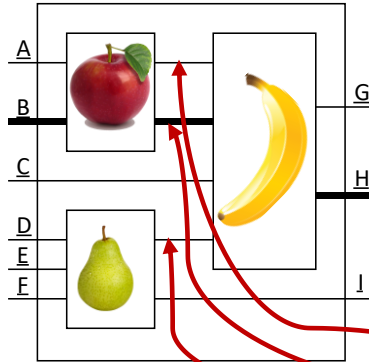
entity D_flipflop is
  port(
    clk: in std_logic;
    D: in std_logic;
    Q: out std_logic
  );
end entity D_flipflop;

architecture data_flow of D_flipflop is
  signal e, f, g, h, i, j, k, l: std_logic;
begin
  -- concurrent statements
  e <= not (D and clk);
  f <= not (e and clk);
  g <= not (e and h);
  h <= not (f and g);
  i <= not (g and not clk);
  j <= not (h and not clk);
  k <= not (l and i);
  l <= not (k and j);
  Q <= k;
end architecture data_flow;

```

# Strukturell kode

- Viser hvordan vi kobler sammen komponenter
- Bruk:
  - Topp- og eller mellom-nivå *når vi har flere moduler*
  - Testbenker



```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
entity my_thing is
port (
  A: in std_logic;
  B: in std_logic_vector(5 downto 0);
  C, D, E, F: in std_logic;
  G: out std_logic;
  H: out std_logic_vector(64 downto 0);
  I: out std_logic
);
end entity my_thing;
```

```
architecture structural of my_thing is
component apple is
port (
  A: in std_logic;
  B: in std_logic_vector(5 downto 0);
  C: out std_logic;
  D: out std_logic_vector(64 downto 0)
);
end component;
```

```
component pear is
port (
  A, B, C: in std_logic;
  D, E: out std_logic
);
end component;
```

```
component banana is
port (
  smurf: in std_logic_vector(64 downto 0);
  cat, dog, donkey: in std_logic;
  horse: out std_logic;
  monkey: out std_logic_vector(64 downto 0)
);
end component;
```

```
signal js: std_logic;
signal ks: std_logic_vector(64 downto 0);
signal ls: std_logic;
```

```
begin
-- port map (component => My_thing)
U1: apple port map(
  A => A,
  B => B,
  C => js,
  D => ks
);
```

```
U2: pear port map(
  A => D,
  B => E,
  C => F,
  D => ls,
  E => I
);
```

```
U3: banana port map(
  smurf => ks,
  cat => js,
  dog => C,
  donkey => ls,
  horse => G,
  monkey => H
);
```

```
end architecture structural;
```

## • Strukturell arkitektur

- Komponentdeklarasjon med porter (ligner entiteter)
- Innvendige signaler
- Koblinger (port map) mellom *komponent* og innvendige *signaler*

# RTL «Register Transfer Level»

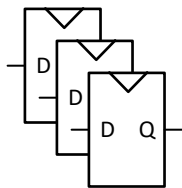
## «Registeroverføringsnivå»

- Høynivå kode

- *hva som skal skje*

- *Ikke hvordan*
    - *Overlater mer til synteseverktøy*

- *bruk av registre*



- *Typisk bruk:*

- *Sekvensielle kretser*

- *Tilstandsmaskiner*
    - *Pipelines*
    - *Tellere*
    - *Etc.*

- Læreboka har en del RTL-beskrivelser

- IN2060: Viktigst å kunne lese RTL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity D_flipflop is
  port(
    clk: in std_logic;
    D: in std_logic;
    Q: out std_logic
  );
end entity D_flipflop;

architecture RTL of D_flipflop is
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      Q <= 'D';
    end if;
  end process;
end architecture RTL;
```

## Oppførsel «Behavioral»

- beskriver hvordan noe skjer i simulering
  - Simuleringsmodeller
  - Testbenker
    - Kobler opp en krets (*strukturelt*)
    - Gir stimuli (input) til kretsen
    - Sjekker output
- *ikke laget for implementering/syntese*

*IN2060 => dataflyt, struktur og enkle testbenker*

# VHDL syntaks

- Typer, porter, signaler og variabler
  - Bruk av porter, retninger
- Vektorer
- Tilordningsoperatorer

# Typer, porter og signaler og variabler

## Typer

- Har definerte verdier og kan ha operasjoner.
- De viktigste typene er definert i egne biblioteker.
  - `std_logic`, `std_logic_vector`

## Porter

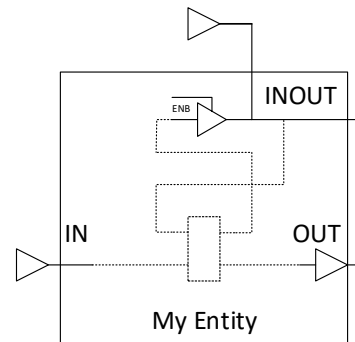
- et "signal" som går inne eller ut av entiteten
- har retning (`in`, `out`, `inout`) og type
  - retningen bestemmer hvor porten tilordnes verdi.

## Signaler

- har type
- tilordnes verdier og leses i arkitekturen

## Variabler

- *har type*
- *Kun for lokal bruk*
  - *innen et statement*  
(dvs innen et subprogram eller en prosess)
- *tilordnes "umiddelbart" med :=*
- *fungerer annerledes enn signaler i noen tilfeller.*



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

entity My_thing is
  port(
    Ain      : in  STD_LOGIC;
    Binout   : inout STD_LOGIC;
    Cout     : out STD_LOGIC);
end entity My_thing;

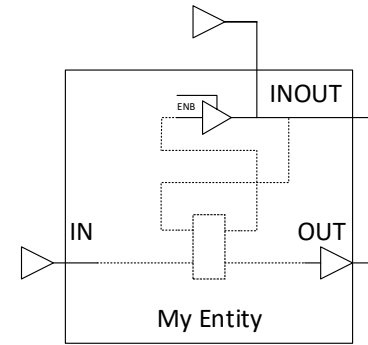
architecture DataFlow of My_thing is
  signal s1, s2, s3, s4 : STD_LOGIC;

begin
  -- ...

  -- concurrent statements
  s1    <= Ain;      -- IN can only drive
  s2    <= Binout;  -- INOUT *can* be driven
  Binout <= s3;     -- INOUT *can* drive
  Cout  <= s4;     -- OUT can only be driven
end architecture DataFlow;
```

# Port - retninger

- "IN"
  - Verdier tilordnes utenfor entiten
  - kan leses, men ikke tilordnes verdier
- "OUT"
  - går ut av entiteten
  - tilordnes verdi i arkitekturen
  - kan også leses som et signal.
- "INOUT" **BANNED**
  - kan gå både inn og ut av entiteten
  - er laget for fysiske tilkoblinger der flere enheter kan drive spenningen til ledning
    - Eks: databussen til et minne, PCI express, etc.
  - **Feil bruk av INOUT gir kortslutninger!**
    - *Kompilatoren vil ikke gi advarsel om flere driver signalet.*
  - IKKE bruk INOUT fordi det er lettvint.



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

entity My_thing is
  port(
    Ain      : in  STD_LOGIC;
    Binout   : inout STD_LOGIC;
    Cout     : out STD_LOGIC);
end entity My_thing;

architecture DataFlow of My_thing is
  signal s1, s2, s3, s4 : STD_LOGIC;

begin
  -- ...

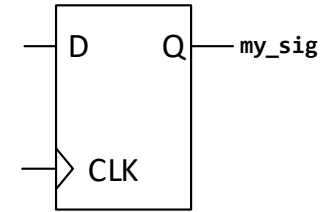
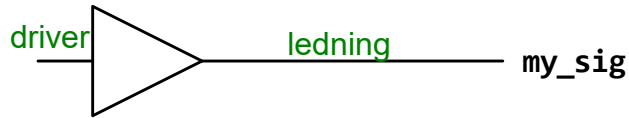
  -- concurrent statements
  s1    <= Ain;
  s2    <= Binout;
  Binout <= s3;
  Cout  <= s4;
end architecture DataFlow;
```



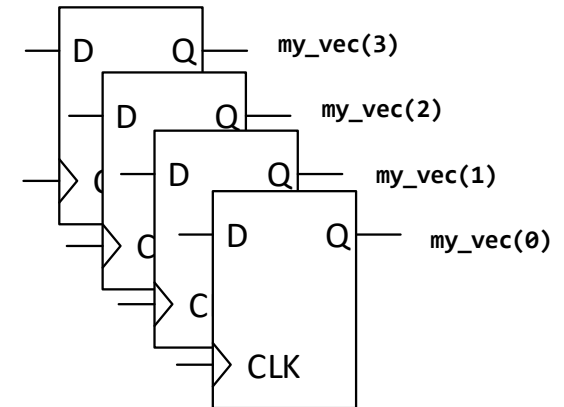
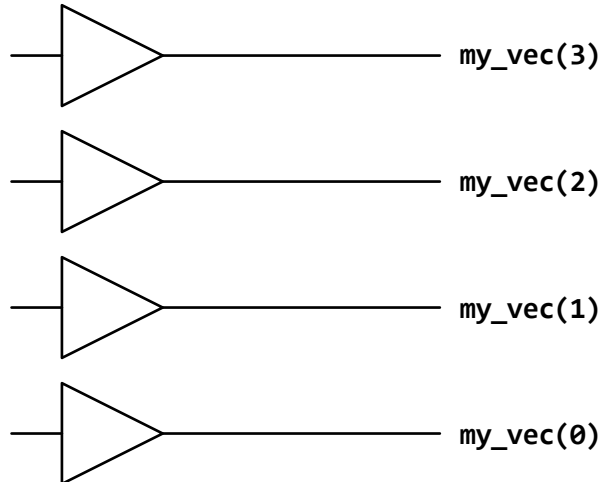
- En vektor er en samling av signaler.
  - Måten vi tilordner verdi til et signal bestemmer om vi får registre eller kombinatoriske kretser.

## vektorer og registre

- `signal my_sig std_logic;`



- `signal my_vec std_logic_vector(3 downto 0);`

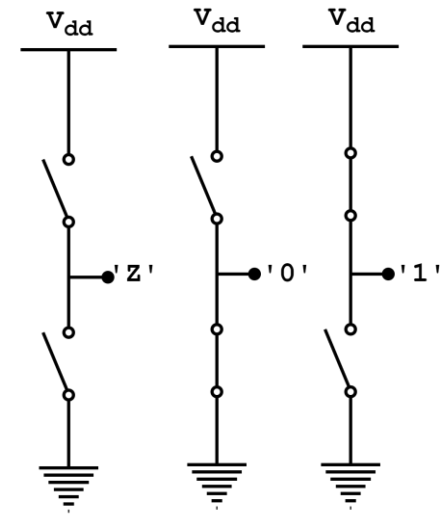
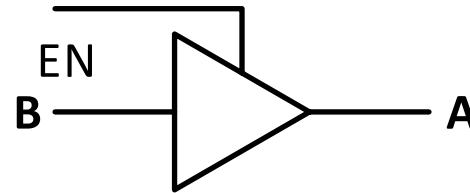


## Tilordning av signaler og variable (assignment)

- `A <= B;` -- A leser B, evt A settes til B, A er et signal
- `C := B;` -- C får verdien av B, C er en variabel  
-- variabler kan bare brukes internt i prosesser, funksjoner og prosedyrer
- `D(6 downto 0) <= E(3 downto 1) & (others => '0');`  
-- D er en vektor med 7 input og  
-- `D(6) <= E(3)`  
-- `D(5) <= E(2)`  
-- `D(4) <= E(1)`  
-- `D(3 downto 0) <= "0000"`
- MERK: vektorer skal ha dobbeltfnutter "00", mens enkeltsignaler skal ha enkeltfnutter '0'

## Tri-state buffer

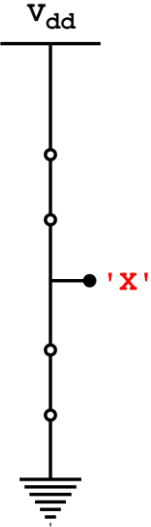
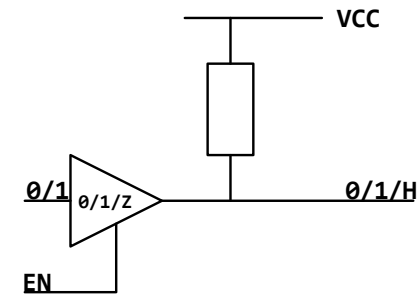
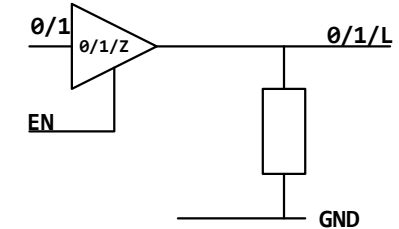
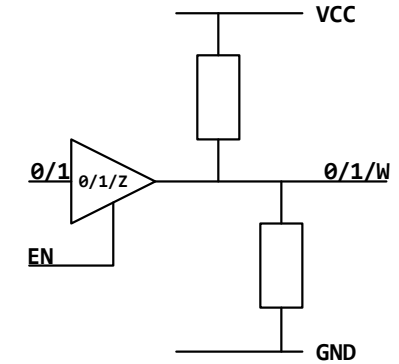
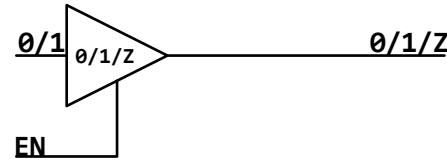
- Digitale kretser kan bare se logisk '0' og '1'
- De kan settes til '0', '1' og 'z' (tristate= høy impedanse)
  - 'z' brukes for at andre skal drive linjen/ bussen uten kortslutning.
- Simuleringsverktøy kan benytte og kontrollere alle de mulige verdiene i **STD\_LOGIC**.



```
A <= B when EN = '1' else 'z';
```

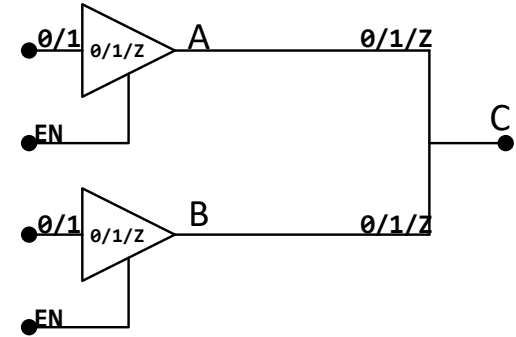
```
-- ekvivalent med  
TRISTATE:  
process (B,EN)  
begin  
  if EN = '1' then  
    A <= B;  
  else  
    A <= 'z';  
  end if;  
end process;
```

# std\_logic -- verdier



Verdi	Betydning	Bruk
'U'	Uninitialized	Startverdi i simulering
'X'	Forcing unknown	Kortslutning, MAO: <b>feil som må fikses</b>
'0'	Forcing zero	Signal drevet lavt, jordet
'1'	Forcing one	Signal drevet høyt
'Z'	High impedance	Signal ikke drevet, tristate
'W'	Weak unknown	Bussterminering (simulering)
'L'	Weak zero	Pull-down motstand (simulering)
'H'	Weak one	Pull-up motstand (simulering)
'-'	Don't care	Used for synthesis and advanced modeling

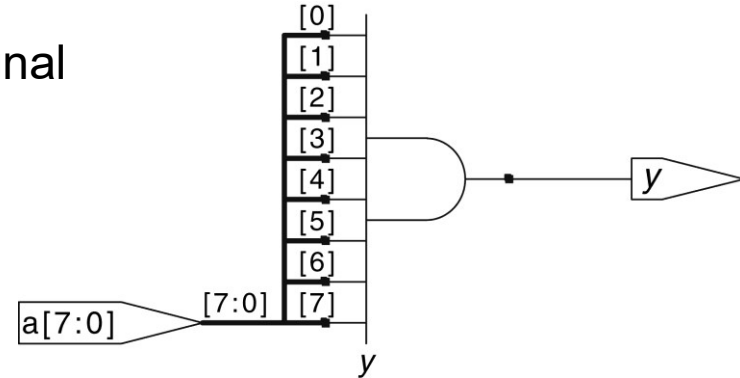
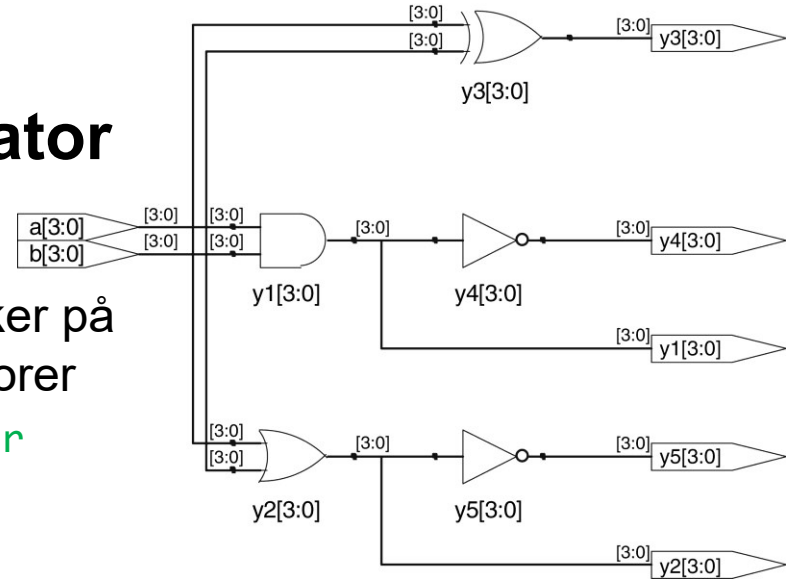
-- multiple drive  
signal c : std\_logic;



Signal A/B	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'_'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'_'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

# Bit operatorer og reduksjonsoperator

- Operatorene **and**, **or**, **nand**, **nor**, **xor**, **xnor** virker på bitnivå når de står mellom to signaler eller vektorer
  - $y1 \leftarrow a \text{ and } b$ ; -- tilsvarer linjene under
    - $y1(3) \leftarrow a(3) \text{ and } b(3)$ ;
    - $y1(2) \leftarrow a(2) \text{ and } b(2)$ ;
    - $y1(1) \leftarrow a(1) \text{ and } b(1)$ ;
    - $y1(0) \leftarrow a(0) \text{ and } b(0)$ ;
- Reduksjonsoperator reduserer en vektor til ett signal
  - $y \leftarrow \text{and } a$  ; -- tilsvarer figur t.h.



# VHDL operator prioritet

- operatorer på samme linje prioriteres slik vi leser; fra venstre mot høyre.
- **Bruk paranteser** for å gjøre det du mener tydelig!

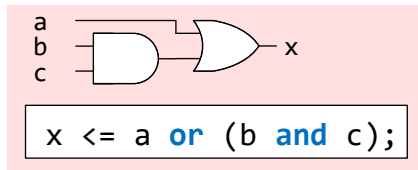
Prioritet	Operator klasse	Operatorer
1 (Utføres først)	miscellaneous	<b>**</b> , <b>abs</b> , <b>not</b>
2	multiplying	<b>*</b> , <b>/</b> , <b>mod</b> , <b>rem</b>
3	sign	<b>+</b> , <b>-</b>
4	adding	<b>+</b> , <b>-</b> , <b>&amp;</b>
5	Shift	<b>sll</b> , <b>sr1</b> , <b>sla</b> , <b>sra</b> , <b>rol</b> , <b>ror</b>
6	relational	<b>=</b> , <b>/=</b> , <b>&lt;</b> , <b>&lt;=</b> , <b>&gt;</b> , <b>&gt;=</b> , <b>?=</b> , <b>?/=</b> , <b>?&lt;</b> , <b>?&lt;=</b> , <b>?&gt;</b> , <b>?&gt;=</b>
7	logical	<b>And</b> , <b>or</b> , <b>nand</b> , <b>nor</b> , <b>xor</b> , <b>xnor</b>
8 (utføres sist)	condition	<b>??</b>

Eksempler: neste slide

# VHDL operator prioritert...

Hva gir

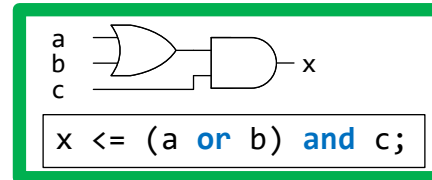
`x <= a or b and c;`



$x = 1 \text{ or } (1 \text{ and } 0)$

$x = 1$

eller



$x = (1 \text{ or } 1) \text{ and } 0$

$x = 0$

Prøv

`a := '1'`

`b := '1'`

`c := '0'`

**Bruk paranteser!**



## Prosesser i VHDL

- Et verktøy for å beskrive kompliserte sammenhenger
  - F.eks sekvensiell logikk (tilstandsmaskiner, pipelines osv.)
  - *Kan brukes til kombinatorisk logikk*
    - Ikke nødvendig i enkle dataflytmodeller
  - Brukes i testbenker
- IN2060: Det som trengs for
  - å forstå kode i læreboka
  - å skrive enkle testbenker
- Senere kurs går i dybden

## Prosesser i VHDL

- To hovedtyper:
  - Med sensitivitetsliste
  - Uten sensitivitetsliste
- *Uten sensitivitetsliste:*
  - Kun til kjøring av testbenk
    - Kan ha innlagt venting "**wait for**"...
  - Gir rekkefølgen ting skal skje i simulering
  - Dette er *nesten* vanlig programmering
    - Utføres i sekvens, men samtidig mellom "**wait**..."

```
STIMULI: process
begin
    wait for 10 ns;

    tb_a <= '1';
    tb_b <= '0';
    tb_cin <= '0';

    wait for 10 ns;

    assert(s = '1')
        report ("Sum ulik 1")
        severity failure;

    tb_a <= '0';
    tb_b <= '1';
    tb_cin <= '1';

    assert(cout = '1')
        report ("feil mente")
        severity failure;

    report("Ferdig!");
    std.env.stop;
end process;
```

## Prosesser med sensitivetsliste

- "ett" statement, én tilordning
  - kan tilordne verdier til flere signaler samtidig
  - beskriver virkemåte til kretser
- Må benyttes til uttrykk som beskrives i rekkefølge
  - `if`, `case`, ...
- Rekkefølgen gir prioritet.
  - Siste tilordning gis høyeste prioritet ved flere tilordninger
  - Dette kan brukes til normalverdier (default-verdier)

```
REGISTERS: process(clk) is
begin
  if rising_edge(clk) then
    reg_A <= neste_A;
    reg_B <= neste_B;
  end if;
end process;
```

```
PRIORITET: process(a,b) is
begin
  -- default tilordning
  y <= '0';
  z <= '0';

  -- betinget tilordning
  if a = '1' then
    if b = '1' then
      y <= '1';
    else
      z <= '1';
    end if;
  end if;
end process;
```

## Prosesser med sensitivetsliste

- Sensitivetslisten (her: `clk`) viser hvilke signaler som *trigger* prosessen
  - Prosessen simuleres bare ved endringer i signalene i listen
- Simuleres i *ett "delta-steg"*.
  - DVS signaler får verdier én gang-
    - *når simuleringen har tolket hele prosessen* (ikke underveis).
  - Rekkefølge gir *prioritet* til signalverdier

```
REGISTERS: process (clk) is
begin
  if rising_edge(clk) then
    reg_A <= neste_A;
    reg_B <= neste_B;
  end if;
end process;
```

```
PRIORITET: process(a,b) is
begin
  -- default tilordning
  y <= '0';
  z <= '0';

  -- betinget tilordning
  if a = '1' then
    if b = '1' then
      y <= '1';
    else
      z <= '1';
    end if;
  end if;

end process;
```

# Prosesser og variable

- Variabler

- *kan* brukes til "mellomregninger"
- *kan ikke* leses utenfor prosess / subprogram
  - Vi må alltid sette signal(er) i en prosess
- Variabler får verdi "*underveis*" med :=
  - *De kan ha mange verdier i én prosess*
    - *Blir som å bruke flere signaler*
  - *Lett å gjøre feil/ tabber*
    - *En variabel som leses før den skrives til gir et lagringselement*
      - » Om y og z kun tilordnes før default-tilordning får vi *latsjer som ikke kan syntetiseres*
      - » Om y og z kun tilordnes før betinget-tilordning blir begge *alltid '0'*

- *I IN 2060 er det viktigst å kunne lese kode med variabler*
  - Begrens bruken til enkle mellomregninger.

```
PRIORITET: process(a,b) is
variable v, w : std_logic;
begin
    -- default tilordning
    v := '0';
    w := '0';

    -- betinget tilordning
    if a = '1' then
        if b = '1' then
            v := '1';
        else
            w := '1';
        end if;
    end if;

    -- tilordning av signaler
    y <= v;
    z <= w;
end process;
```

# Testbenker

- Testbenker er moduler som brukes til å gi generere testdata (testvektorer) og rapportere testresultater ved simulering.
  - De syntetiseres ikke
    - *er prinsipielt software, selv om det er skrevet med et HDL*
- Kan bruke én eller flere moduler som komponenter.
  - Vi tester typisk én modul av gangen
    - vi navngir denne «DUT» eller «UUT» (Device / Unit under test).
  - Øvrige komponenter er med for å gi simuleringsdata
- NB: *Testbenk-kode vil ligne mer på andre programmeringsspråk,*
  - *Fordi tester har sekvensielle hendelser*
  - *Vi kan gjøre I/O til skjerm og fil*
  - Vi kan teste ulike ting samtidig i simulering (i parallell)
    - Selv om stimuli settes i sekvens
  - *Kan være forvirrende..*

## Simuleringsprosess og wait statement

- I en testbenk kan vi ha *én prosess uten sensitivetsliste*
  - Denne prosessen brukes til å sette stimuli i sekvens
    - Vi kan ikke ha flere slike prosesser (=> umulig å simulere)
  - I den prosessen kan vi bruke "wait" for å vente ulike tidsrom
    - `wait for 1 ns;`
    - `wait until b = '1';`
- `wait until` kan også benyttes utenfor testbenker (*ikke tema i IN2060*)

## Tidsforsinkelser (i simulering/testbenker)

- I simuleringsprosessen kan man bruke `wait for`:
  - `wait for 5 ns;`
- `type TIME is range implementation_defined`  
`units`
  - `fs; -- femtosecond (the smallest predefined time-step in VHDL simulation).`
  - `ps = 1000 fs; -- 1 picosecond = 1000 femtoseconds... etc`
  - `ns = 1000 ps; -- nanosecond`
  - `us = 1000 ns; -- microsecond`
  - `ms = 1000 us; -- millisecond`
  - `sec = 1000 ms; -- second`
  - `min = 60 sec; -- minute`
  - `hr = 60 min; -- hour``end units;`



## assert – test om noe feiler

- `assert(a = b) report("min rapport") severity note;`
  - Hvis *ikke* a er lik b, så leveres "min rapport" til skjerm som «note» .
- Severity kan være
  - `note` -- ufarlig, typisk melding om at en del av testingen er gjennomført
  - `warning` -- brukes typisk til meldinger om noe uventet
  - `error` -- (default) brukes til feil, men testing fortsetter
  - `failure` -- brukes til showstoppere- stopper simuleringen
- `report();` -- kan benyttes alene (rapporterer alltid)
  - `severity` er `note` om ikke annet angis.

# Testbenk eksempel

- Tom entitet
- Komponenter er definert som entitet i en annen fil
- Signaler i testbenken brukes som input til komponentene vi tester
- Vi instansierer komponenter som strukturell kode
- Stimuli gis i fra en eller flere prosesser
- Vi bruker assert + report for å gi beskjeder og feilmeldinger
- Std.env.stop stopper simuleringen (ellers starter prosessen på ny)

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

```
entity tb_fulladder is  
end entity tb_fulladder;
```

```
architecture behavioral of tb_fulladder is
```

```
component fulladder is  
port(  
a, b : in std_logic;  
cin : in std_logic;  
s : out std_logic;  
cout : out std_logic  
);  
end component;
```

```
signal tb_a, tb_b, tb_cin : std_logic := '0';  
signal tb_s, tb_cout : std_logic;
```

```
begin
```

```
DUT: fulladder  
port map(  
a => tb_a,  
b => tb_b,  
cin => tb_cin,  
s => tb_s,  
cout => tb_cout  
);
```

```
STIMULI: process
```

```
begin
```

```
wait for 10 ns;  
tb_a <= '1';  
tb_b <= '0';  
tb_cin <= '0';  
wait for 10 ns;
```

```
assert(s = '1') report ("Sum ulik 1") severity failure;
```

```
tb_a <= '0';  
tb_b <= '1';  
tb_cin <= '1';
```

```
assert(cout = '1') report ("feil mente") severity failure;  
report("Ferdig!");
```

```
std.env.stop;
```

```
end process;
```

```
end architecture behavioral;
```

## Oppsummeringsspørsmål:

- Hva er formålene med et hardwarespråk?
  1. *Definere digitale kretser*
  2. *Teste (simulere) kretser*
- Hva er forskjellen på en entitet og en arkitektur?
  - *Entitet viser innganger og utganger på designet*
  - *Arkitektur beskriver funksjonen til innmaten.*
- Hvorfor trenger vi testbenker?
  - *Fordi alle gjør feil nesten hele tiden, og det gjelder å finne dem tidlig.*

# Oblig i digital design, CLA-adder:

- Kjøre og modifisere testbenk med skriving til konsoll (skjerm)
- Lage dataflyt kode
- Strukturell kode
- *Obligen tar dere gjennom design av én type carry-lookahead adder (CLA)*
  - *Den kan hjelpe til å forstå hvordan vi bygger slike kretser*
    - *Det er helt greit å starte på obligen før man forstår CLA fullt ut*
  - For å få godkjent:
    - *Kompilerbar kode*
    - *Gjennomført simuleringer*
    - *Relevante tidsdiagrammer (waveforms)*
    - *Vurdere om resultatet er rimelig (kort)*
    - *Ha en rimelig ide om mentepropagering i en CLA adder*
- **Anbefaling:**
  - Ta **ukesoppgavene** for denne uken **først**
  - *Ikke sitt alene* med obligen siste kveld!
    - Det er hver enkelts ansvar å ta kontakt om noe er vanskelig
    - *Bruk*
      - gruppetimer/gruppelærerne
      - medstudenter
      - kommunikasjonsplattform (discourse)

# Ukeoppgaver

## Oppgaver

**4.1**, 4.3, **4.5**, **4.6**, 4.8, 4.9, 4.10  
(ekstra 4.20, 4.22)

For 4.6, se figur 2.47 og 2.48 s 79, 80.

Der dere skal skrive HDL-kode, kan denne åpnes og simuleres med Questa. *Oblig 1 forteller hvordan.*

# VHDL Number format (kun til orientering)

Binary, Decimal, hexadecimal, Octal

Unsigned, Signed

<ant bit><U/S><B/D/O/X> "<tall av type B/D/O/X>"

**B**"1111\_1111\_1111" -- Equivalent to the string literal "111111111111".

**X**"FFF" -- Equivalent to **B**"1111\_1111\_1111".

**O**"777" -- Equivalent to **B**"111\_111\_111".

**X**"777" -- Equivalent to **B**"0111\_0111\_0111".

**B**"XXXX\_01LH" -- Equivalent to the string literal "XXXX01LH"

**U0**"27" -- Equivalent to **B**"010\_111"

**U0**"2C" -- Equivalent to **B**"011\_CCC"

**SX**"3W" -- Equivalent to **B**"0011\_WWWW"

**D**"35" -- Equivalent to **B**"100011"

**12UB**"X1" -- Equivalent to **B**"0000\_0000\_00X1"

**12SB**"X1" -- Equivalent to **B**"XXXX\_XXXX\_XXX1"

**12UX**"F-" -- Equivalent to **B**"0000\_1111\_----"

**12SX**"F-" -- Equivalent to **B**"1111\_1111\_----"

**12D**"13" -- Equivalent to **B**"0000\_0000\_1101"

**12UX**"000WWW" -- Equivalent to **B**"WWW\_WWWW\_WWWW"

**12SX**"FFFC00" -- Equivalent to **B**"1100\_0000\_0000"

**12SX**"XXXX00" -- Equivalent to **B**"XXXX\_0000\_0000"

**8D**"511" -- Error

**8U0**"477" -- Error

**8SX**"0FF" -- Error

**8SX**"FXX" -- Error

# VHDL bokstavstørrelse (kun til orientering)

- *til forskjell fra enkelte andre språk...*
  - ...eR iKke vHdL cAsE sENsItiV
  - å oVERDRivE brUKEn AV dEnnE
  - FRihETeN eR IKKE Å anBEfale.
- Tips:
  - ALL\_CAPS BLIR SLITSOMT NÅR DET ER FOR MYE AV DET!
  - For VHDL anbefales *snake\_case* til vanlig og *ALL\_CAPS* for konstanter