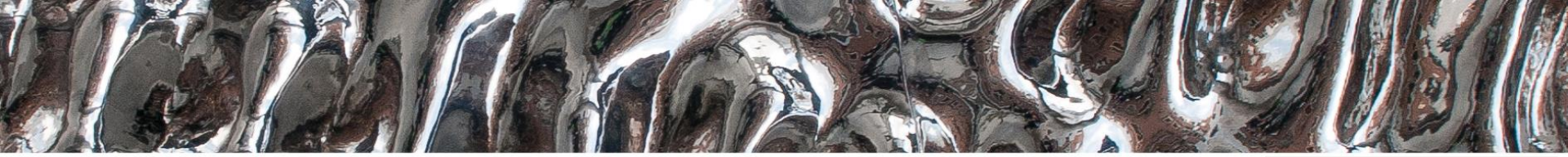
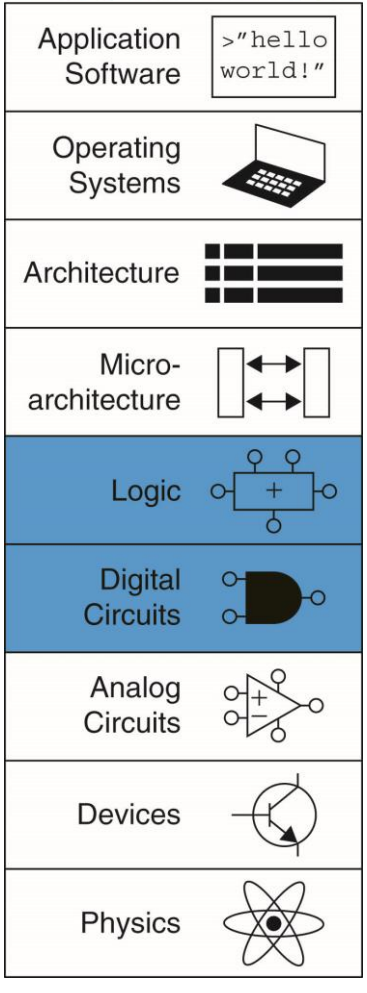


UiO : Department of Informatics
University of Oslo

F4 IN2060 2022

HDL

Yngve Hafting, yngveha@ifi.uio.no

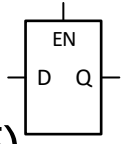


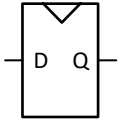
Formål

- **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)**
    - **Demonstrasjon av verktøy**
    - **HDL eksempler, digitale byggeblokker**
    - **Gjennomgang av oppgaver**

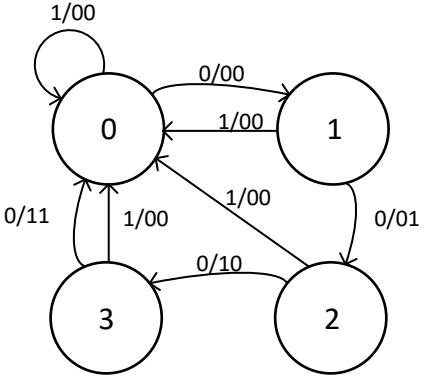
Fra forrige gang

- Forskjellige «vipper» latch og flippflopp:
  - Latch:
    - Nivå
  - Flippflopp (FF)
    - Flanke
  - 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



Hvor finner vi digitale design i dag?

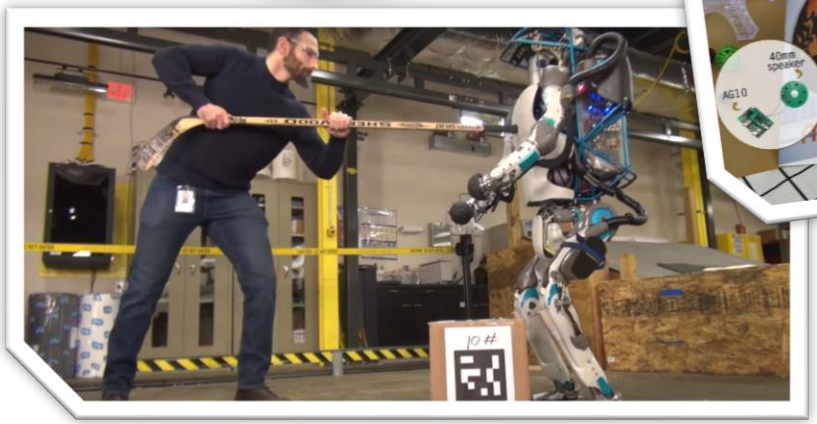
- ...

Hvor finner vi digitale design i dag?

- Ting!
  - Gjenstander som har en form for elektronisk kontroll.

- ≠ Programvare...
  - **HDL** = *Hardware Description Language*
    - **definerer virkemåten til digitale kretser.**
      - *(og simulering av disse)*
      - *designet blir **ikke** kjørt **sekvensielt** som et program.*

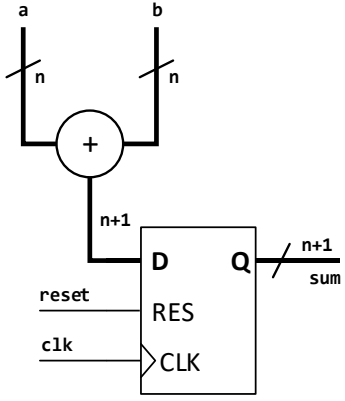
UiO : Department of Informatics
University of Oslo



https://www.youtube.com/watch?v=rVlhMGQgDkY

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»

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

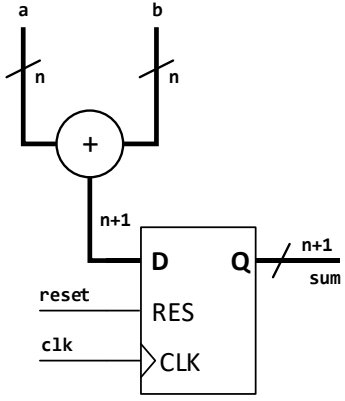
```
MOV R5, #0       ;set base adr
LDR R7, [R5, #8] ;load reg R7
LDR R8, [R5, #12];load reg R8
ADD R0, R7, R8   ;R0=R7+R8
STR R0, [R5, #16];store R0
```

```
01001100 10011001 00100100 01001010
11001100 10111001 01100100 11110110
01001100 10011001 00111100 11101010
01001100 10011001 00100100 01011010
…
```

(binærkoden tilfeldig sammensatt for illustrasjonsformål)

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 **logisk funksjon** til en **krets**	Definerer **instruksjoner** og deres rekkefølge for en **prosessor**
CAD (DAK) verktøy **syntetiserer** designet slik at det kan realiseres med fysiske porter.	***En kompilator oversetter programmet til maskinkode*** som prosessoren kan kjøre sekvensielt fra minne.
Implementeres* i *programmerbar logikk (**PL**) eller *applikasjons-spesifikke kretser* (**ASIC**) (*PL* = FPGA, CPLD, PLD, PAL, PLA, …) (ASIC = hva-som-helst-slags-chip: prosessorer, mikrokontrollere, osv.)	**Lagres i minnet til en datamaskin**
Verilog (SystemVerilog) **VHDL (VHDL 2008)** (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)

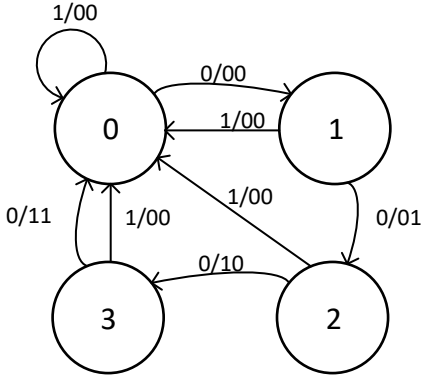
Designprosess

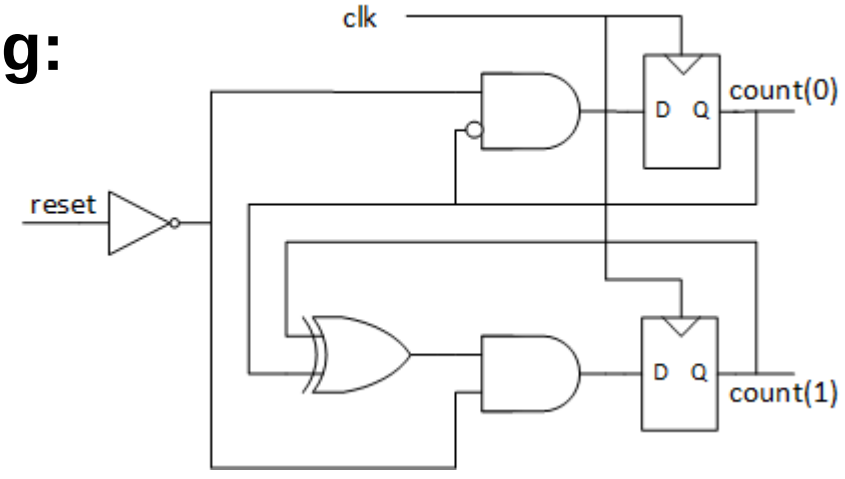
- Noen flere step
- Mer bruk av verktøy
- Simulering kan gjøres for hvert nye steg
  (Etter syntese, portplassering og implementering)

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

HDL vs skjemategning:

- Eks: 2 bit teller



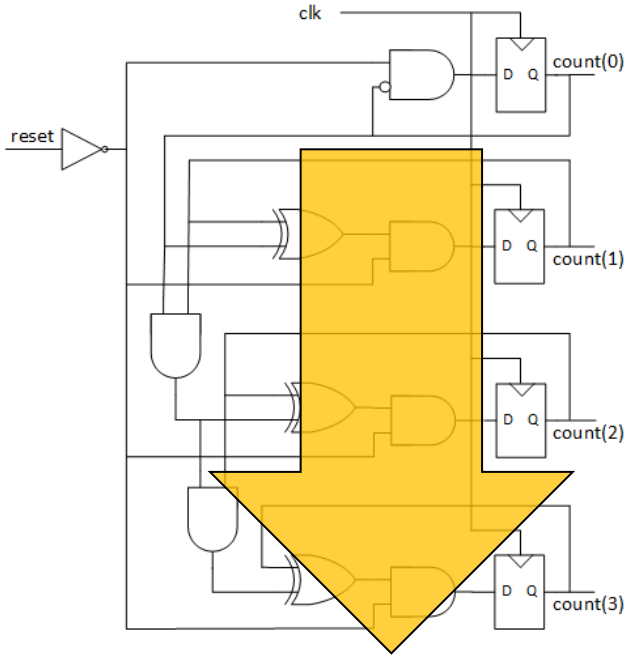



```
generic (N : positive := 2);
---
count : out std_logic_vector (N-1 downto 0);
---
signal next_count : std_logic_vector (N-1 downto 0);
---
-- register assignment
count <= next_count when rising_edge(clk);

-- 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 );
```

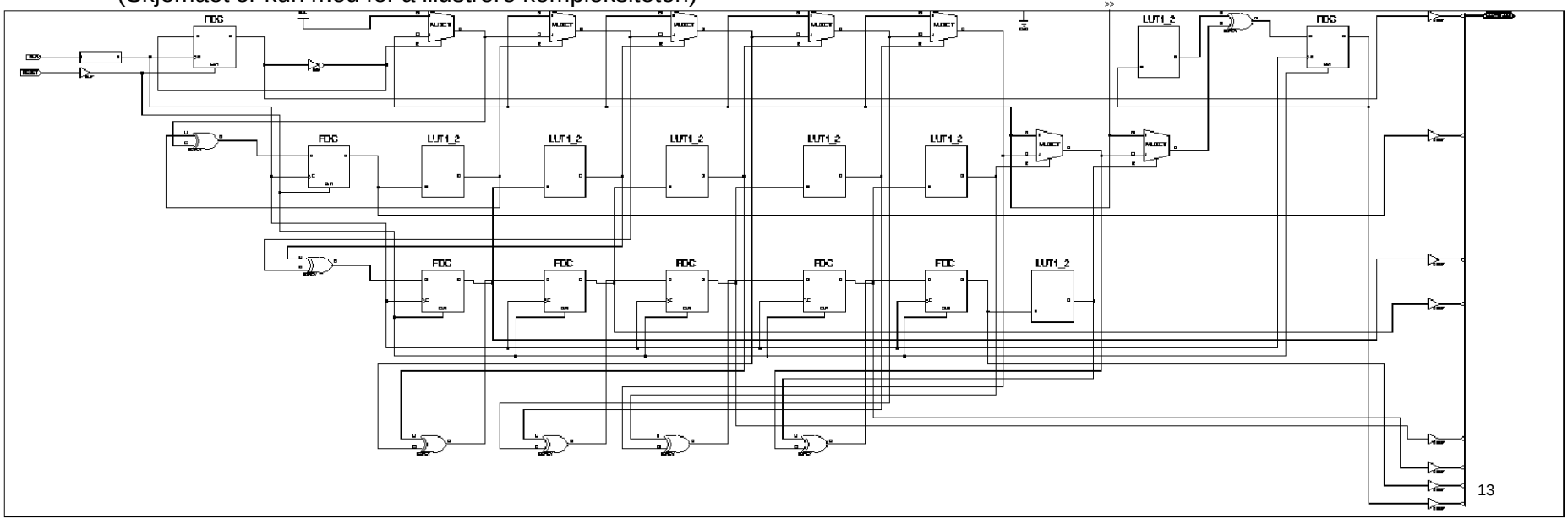
8 bit, …

```
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 );
```

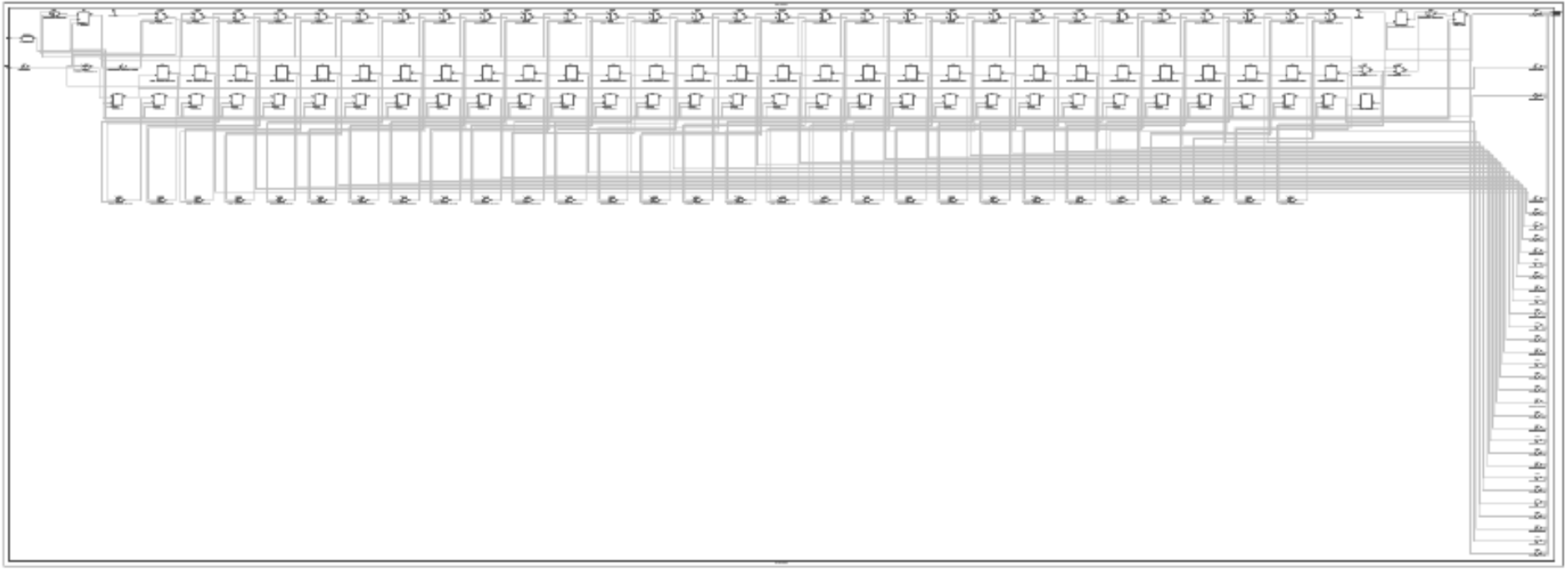
Autogenerert skjema for implementasjon fra synteseverktøy

- 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…

- Lære enkel bruk av VHDL til simulering
  - *Koding av kombinatoriske kretser*
  - *Simulering med testbenk*
- VHDL brukes til å forklare funksjon av datamaskinarkitektur
  - (Vi bruker ikke systemVerilog)

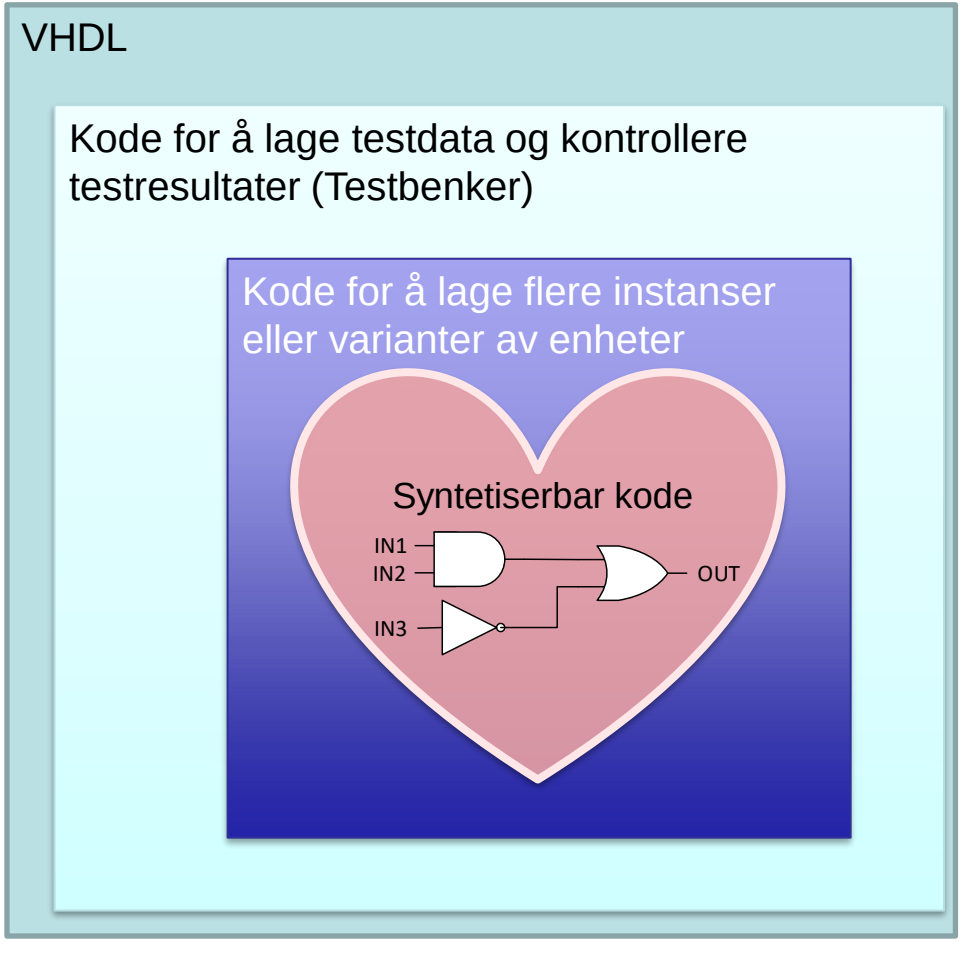
Senere kurs (IN3160)…

- *realisere fysiske kretser* (programmerbar logikk)
- Mer om VHDL
  - *Mer syntaks*
  - *Subprogram*
  - *…*
- 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
  - **det som beskrives er en krets, *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..!*



Krasjkurs i VHDL..:

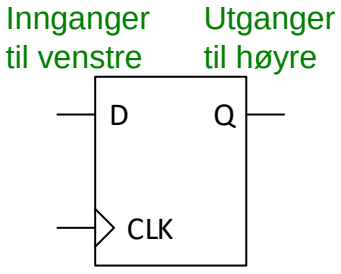
- Oppbygging
  - Entitet og arkitektur
- Kodetyper
  - dataflow
  - structural
  - RTL
  - behavioral
- Syntaks og konkret koding av design
  - Datatyper og verdier
  - Operatorer
  - Tilordning
  - Duplisering av elementer
- Simulering
  - Testbenker

VHDL oppbygging

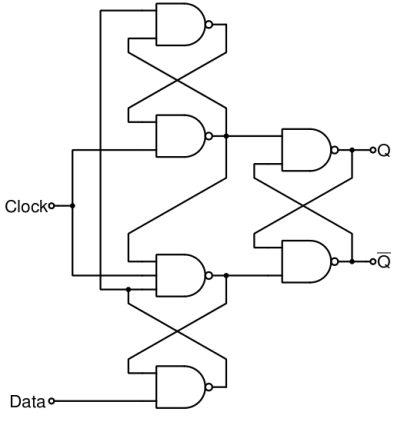
Entitet og arkitektur

- Entiteten definerer input og output

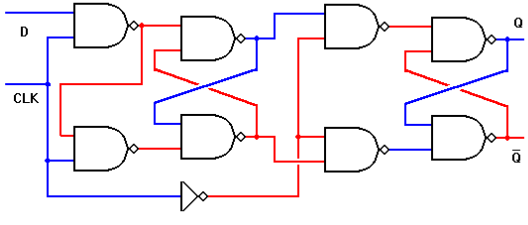
- Arkitekturen beskriver hva designet gjør.
  - En entitet kan benytte forskjellige arkitekturer (kun én av gangen)
  - Arkitekturer kan defineres med forskjellige stiler (neste slide)
    - "RTL" og "Dataflow" er navn på kodestil.
      - (mer følger…)




```
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;
```



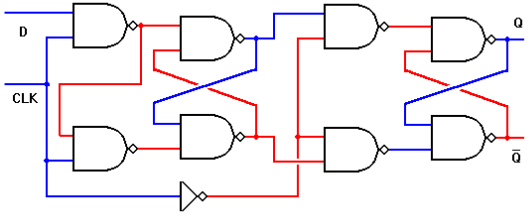

```
architecture data_flow of D_FLIPFLOP is
 signal e,f,g,h,i,j,k,l : std_logic;
begin
  e <= NOT (D and clk);
  f <= NOT (e and clk);
  g <= NOT (e and h);
  h <= NOT (f and g);
  i <= NOT (g and not clk);
  j <= NOT (h and not clk);
  k <= NOT (l and i);
  l <= NOT (k and j);
  Q <= k;
end architecture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

4 Kodetyper / stiler:

- Dataflyt “data_flow”
- Strukturell “structural”
- RTL “Register transfer level”
- Behavioral

Dataflyt

- Beskriver 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

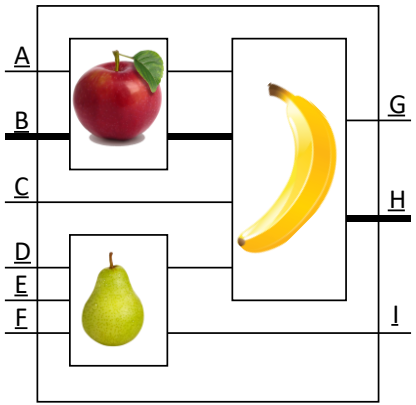



```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

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_t
  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 (ligner entiteter)
  - Innvendige signaler
  - Koblinger (port map) mellom *komponent* og innvendige *signaler*

RTL «Register Transfer Level»

- Høynivå kode
  - *hva som skal skje*
    - *Ikke hvordan*
    - Overlater mer til synteseverktøy
  - *bruk av registre*

- *Typisk bruk:*
  - Sekvensielle kretser
    - *Tilstandsmaskiner*
    - *Pipelines*
    - *Tellere*
    - *Etc.*
- IN3160 tar for seg 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»

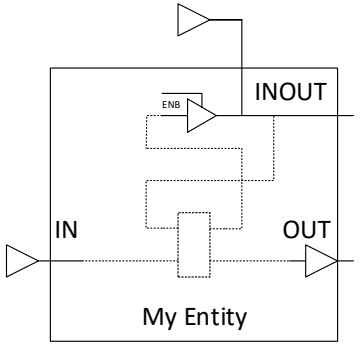
- beskriver hvordan noe skjer i simulering
  - Simuleringsmodeller
  - Testbenker
    - Kobler opp en krets *(strukturelt)*
    - Gir stimuli (input) til kretsen
    - Sjekker output
- *ikke laget for implementering/syntese*

IN2060 => dataflyt, struktur og enkle tesbenker

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

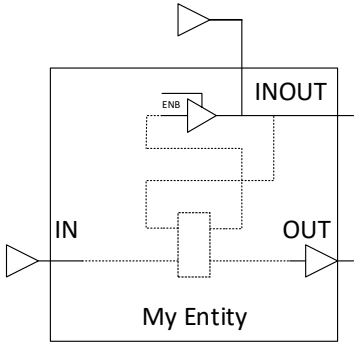
- har type
- tilordnes verdier og leses i arkitekturen

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



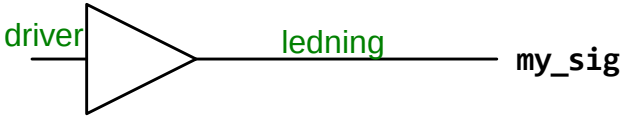
- "IN"
  - settes utenfor entiten
  - kan leses, men ikke tilordnes verdier
- "OUT"
  - går ut av entiteten
  - tilordnes verdi i arkitekturen
  - kan også leses som et signal.
- "INOUT"
  - kan gå både inn og ut av entiteten
  - er laget for fysiske tilkoblinger der flere enheter kan drive spenningen 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.

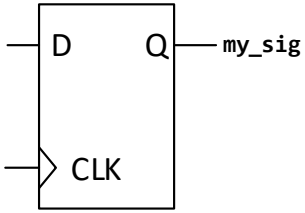

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

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.*

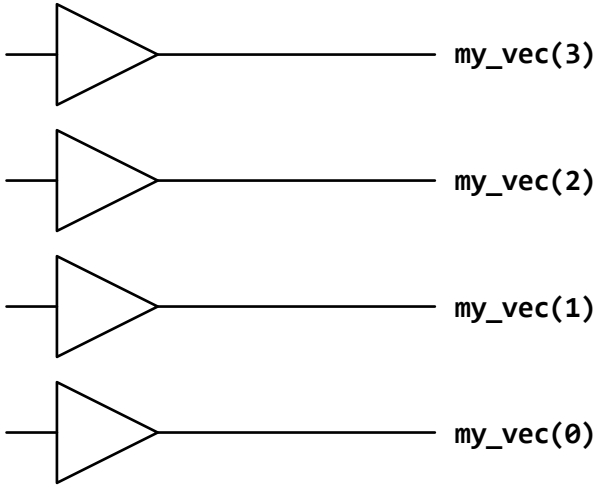
vektorer og registre

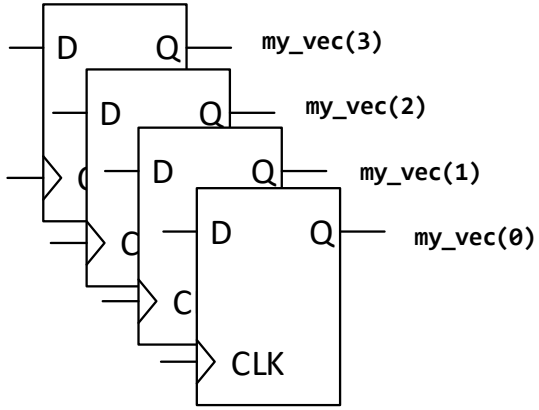
- `signal my_sig std_logic;`





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



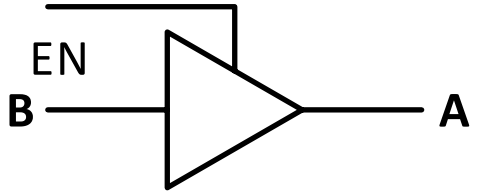


Tilordning av signaler og variable (assignment)

- `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

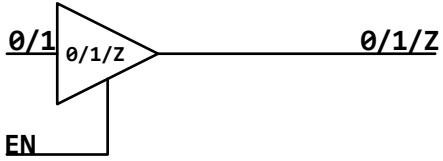


- Digitale kretser kan bare se logisk ‘0’ og ‘1’
- De kan settes til ‘0’, ‘1’ og ‘z’ (tristate= høy impedanse)
  - ‘z’ brukes for at andre skal drive linjen/ bussen uten kortslutning.

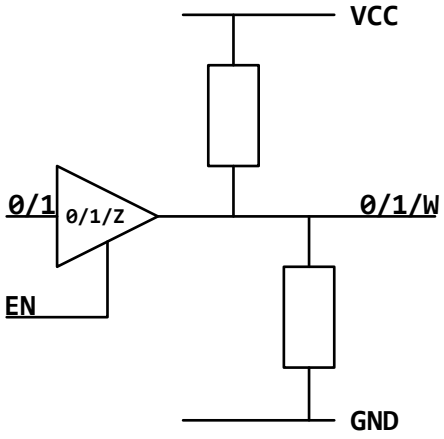
- Simuleringsverktøy kan benytte og kontrollere alle de mulige verdiene i `STD_LOGIC`.

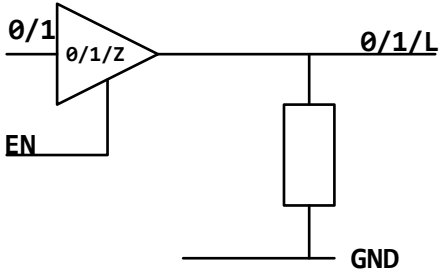
```
A <= B when EN = '1' else 'Z';

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

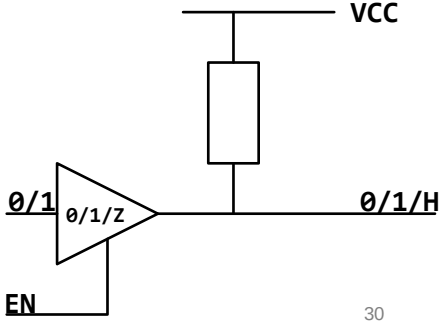
std_logic -- verdier



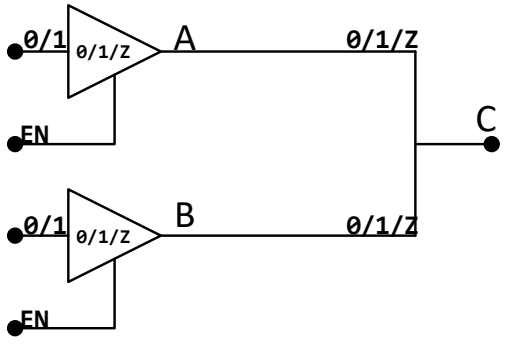
Value	Name	Usage
‘U’	Uninitialized state	Used as a default value
'X'	Forcing unknown	Bus contentions, error conditions, etc.
‘0’	Forcing zero	Transistor driven to GND
‘1’	Forcing one	Transistor driven to VCC
‘Z’	High impedance	3-state buffer outputs
‘W’	Weak unknown	Bus terminators
‘L’	Weak zero	Pull down resistors
‘H’	Weak one	Pull up resistors
‘-’	Don’t care	Used for synthesis and advanced modeling





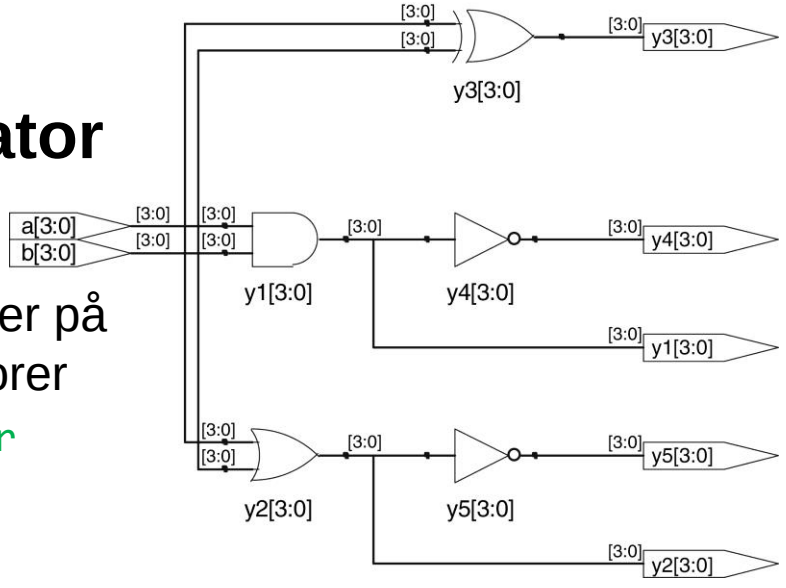


```
-- multiple drivere
signal c : std_logic;
```



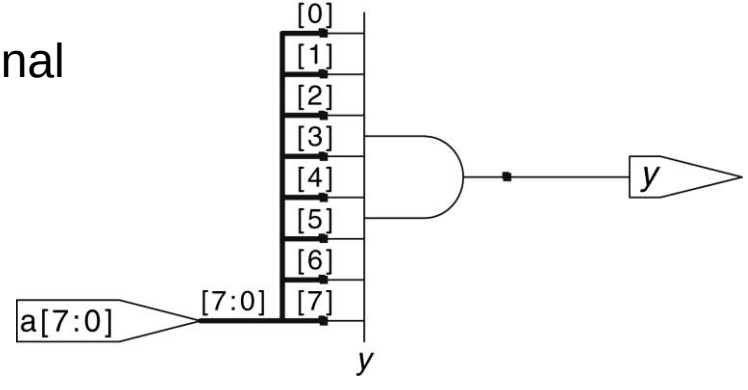
Signal A/B	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'–'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'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 <= a and b; -- tilsvarer linjene under`
    - `y1(3) <= a(3) and b(3);`
    - `y1(2) <= a(2) and b(2);`
    - `y1(1) <= a(1) and b(1);`
    - `y1(0) <= a(0) and b(0);`

- Reduksjonsoperator reduserer en vektor til ett signal
  - `y <= and a ; -- tilsvarer figur t.h.`



VHDL opreator prioritet

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

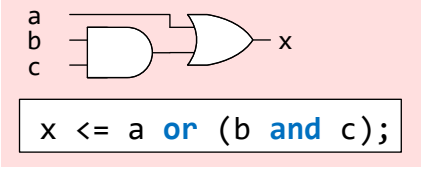
Prioritet	Operator klasse	Operatorer
1 (Utføres først)	miscellaneous	`**, abs, not`
2	multiplying	`*, /, mod, rem`
3	sign	`+, -`
4	adding	`+, -, &`
5	Shift	`sll, srl, sla, sra, rol, ror`
6	relational	`=, /=, <, <=, >, >=, ?=, ?/=, ?<, ?<=, ?>, ?>=`
7	logical	`And, or, nand, nor, xor, xnor`
8 (utføres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

Eksempler: neste slide

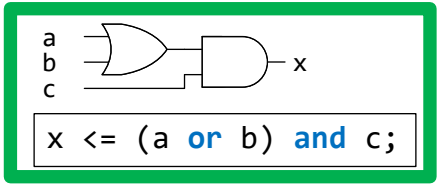
VHDL operator prioritet 1/2

Hva gir

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



eller



Prøv
a := '1'
b := '1'
c := '0'

x = 1 or (1 and 0)

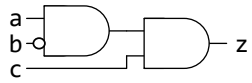
x = 1

x = (1 or 1) and 0

x = 0

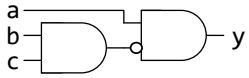
VHDL operator prioritet 2/2

```
z <= a and not b and c; ==
z <= a and (not b) and c; ==
z <= (a and (not b)) and c;
```




```
-- z=1 only when a=1, b=0, c=1.
```

```
y <= a and not (b and c);
```




```
--  y=1 when a=1, b=0 or when a=1, c=0.
```

Bruk paranteser…

Prosesser i VHDL

- En prosess- `process()` -er *ett statement*...
  - brukes når tilordningsregelen blir kompleks
    - *Kan tilordne flere signaler på en gang*
    - *Kan tilordne basert på prioritet (= fallgruve)*
    - *Kan ha variable som brukes lokalt*
  - `case` og `if` må ha `process()`
    - *Fordi disse kan sette flere signaler*
  - Signaler settes kun én gang per gjennomkjøring
    - *i det vi går ut av prosessen*
- `process(`<***Sensitivitets liste***>`)`
  - prosesser trigges når ett av signalene i listen endres
    - Simulering *må ha* sensitivitetsliste
    - Synteseverktøy *ignorerer* listen… *men klager om den mangler*
- Prosesser har én driver per signal
  - Å tilordne flere drivere til samme signal innen en prosess medfører *prioritering.*
    - **Siste- overstyrer** tidligere tilordninger.
    - *Utenfor prosess er det ikke lov… (= kortslutning)*

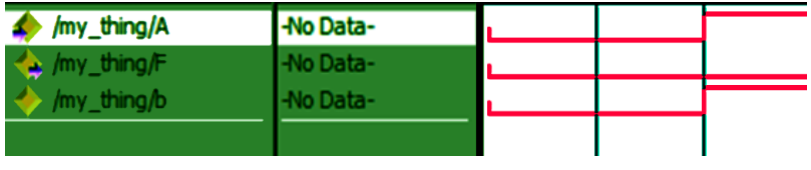
- Anta b = 0,
  A går fra 0 til 1.
- Hva skjer med F?
- Hvorfor?

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

entity My_thing is
    port(A: in STD_LOGIC;
         F: out STD_LOGIC
    );
end entity My_thing;


architecture Behavioral of My_thing is
    signal b : STD_LOGIC;
begin
    signal_update: process(a)
    begin
        if A = '1' then b <= '1';
        else b <= '0';
        end if;

        if b = '1' then F <= '1';
        else F <= '0';
        end if;

    end process;
end architecture Behavioral;
```

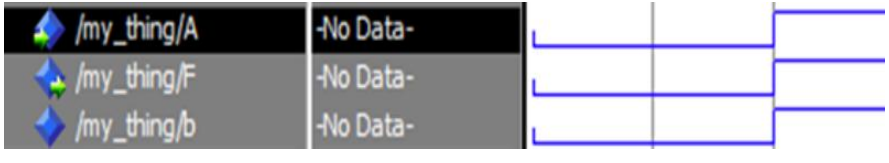
Sensitivitetsliste

- Hva skjer med F når A går fra 0 til 1?

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

entity My_thing is
    port(A: in STD_LOGIC;
         F: out STD_LOGIC
    );
end entity My_thing;


architecture Behavioral of My_thing is
    signal b : STD_LOGIC;
begin
    signal_update: process(a)
    begin
        if A = '1' then b <= '1';
        else b <= '0';
        end if;

        if b = '1' then F <= '1';
        else F <= '0';
        end if;

    end process;
end architecture Behavioral;
```



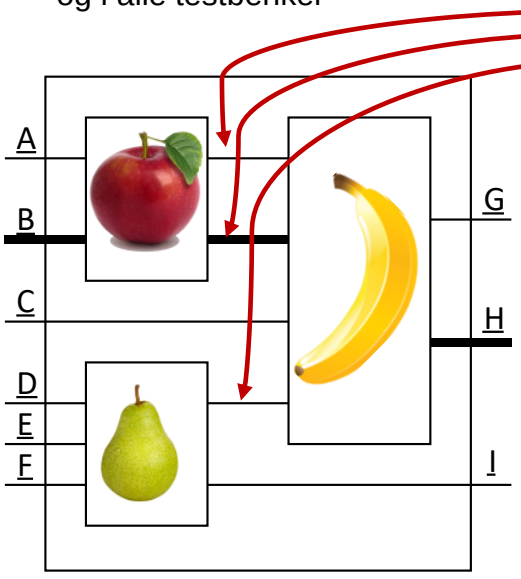

```
signal_update: process(a,b)
begin
  if a = '1' then
    b <= '1';
  else
    b <= '0';
  end if;
  if b = '1' then
    f <= '1';
  else
    f <= '0';
  end if;
end process;
```



Strukturell kode

Strukturell

- Hvordan vi kobler sammen eksisterende komponenter
- Brukes typisk i større design og i alle 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
  signal js: std_logic;
  signal ks: std_logic_vector(64 downto 0);
  signal ls: std_logic;

  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;

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;
```

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 sensitivitetsliste*
  - 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
        ps = 1000 fs; -- picosecond
        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 hardware*
  2. *Teste (simulere) hardwaredesign*
- 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:

- Oppgave 1
  - Kjøre og modifisere en testbenk
- Oppgave 2, CLA-blokk med testbenk
  - Dataflyt kode
  - Testbenk med skriving til konsoll (skjerm)
- Oppgave 3 CLA-adder toppmodul
  - Strukturell kode
  - Testbenk

- *Obligen tar dere gjennom design av en type carry-lookahead adder (CLA)*
  - *Den er kan hjelpe til å forstå hvordan vi bygger slike kretser*
    - *Det er helt greit å starte på obligen før man forstår CLA fullt ut*

- *Anbefaling:*
  - *Gjør ukesoppgavene for denne uken først*

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, **D**ecimal, he**X**adecimal, **O**ctal
Unsigned, **S**igned
<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"
UO"27" -- Equivalent to B"010_111"
UO"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"WWWW_WWWW_WWWW"
12SX"FFFC00" -- Equivalent to B"1100_0000_0000"
12SX"XXXX00" -- Equivalent to B"XXXX_0000_0000"
8D"511" -- Error
8UO"477" -- Error
8SX"0FF" -- Error
8SX"FXX" -- Error
```

VHDL bokstavstørrelse (kun til orientering)

- *til forskjell fra SystemVerilog o.a...*

  …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