# Design og simulering av 32 bits carry-lookahead-adder med dataflyt-VHDL

*En oppgave i å konstruere og analysere digitale kretser på lavnivå.*

Av Yngve Hafting 2020

## CLA-adder

CLA er en forkortelse for carry-lookahead-adder. CLA addere vil ha lavere propagerinsforsinkelse enn ripple-carry-addere som er den enkleste formen for multi-bit-addere.  I kurset vil dere lære mer om hvordan disse virker, når vi kommer til kapittel 5 i læreboka. Det er ikke nødvendig å

## Dataflyt-, RTL-, og strukturell-kode

En dataflytbeskrivelse beskriver hvordan signaler bearbeides på port-nivå (med logiske porter som AND, OR, XOR, osv). Strukturell kode beskriver hvordan moduler settes sammen. RTL kode beskriver hva som skjer fra register til register, men spesifiserer gjerne ikke i detalj hvordan noe skal skje- det gjør man med dataflyt-kode. Dataflytkode kan inngå i RTL-kode, og strukturell kode kan danne dataflyt kode, men i praksis er det fornuftig å skille disse, og være bevisst hva man jobber med.

Når vi skriver dataflyt-kode (her med VHDL), har vi størst mulig kontroll på hvilke porter som brukes, og vi styrer i detalj hvor optimal løsningen blir - på godt og vondt. Skal man kunne ta stilling til når det er lurt å benytte dataflyt fremfor høynivåkode, må man ha mer inngående kjennskap til digital design enn målet for dette kurset. Kurset IN3160 - Digital systemkonstruksjon, tar dette videre. I dette kurset er målet først å få kontroll på den grunnleggende dataflyten og strukturen.

## Simulering og testbenk

Å få større digitale design til å virke er tilnærmet umulig uten testing, og i en designprosess tar testingen ofte større plass enn designet av kretsen man skal implementere. Testing kan gjøres på mange måter og nivåer, med varierende grad av automasjon. På det enkleste nivået så laster man en modul inn i en simulator og bruker det grafiske grensesnittet i simulatoren til å sette ett signal av gangen før man ber den kjøre en tidsperiode. Denne prosessen kan gjentas til det kjedsommelige for å teste en modul fullstendig, men det er svært tungvint og ta mye tid. For å kjøre en rekke tester automatisk, kan man lage en «testbenk» som genererer inputene for oss. For de fleste praktiske formål, vil det være mer effektivt å benytte en testbenk enn å manipulere signaler enkeltvis ved simulering. Testbenken kan gjenbrukes selv om innmaten i en modul endres på, så lenge portene definert i entiteten er de samme.

Testbenker i VHDL kan lages selvsjekkende, med rapportering til skjerm og fil, samt bruk av testvektorer man kan ha lagret på filer i varierende format. Man kan også lage tester med skriptspråk som TCL, eller ved bruk av andre programmeringsspråk som for eksempel Python, men det vil i stor grad være verktøyavhengig om slike løsninger kan benyttes.

*I dette kurset er det først og fremst et mål å kunne lage enkle testbenker for å teste små moduler laget med dataflyt-VHDL. I denne obligen, skal du skrive testbenker som manipulerer input og sjekker om output er riktig. I tillegg skal du benytte det grafiske grensesnittet til simuleringsverktøyet til å sjekke resultatet.*

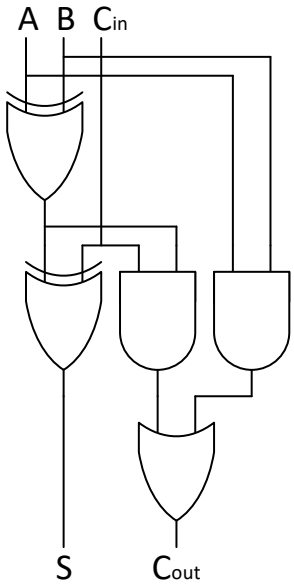
## Målet med obligen

I denne obligen er målet å få øvelse i å lage egen dataflyt kode i VHDL, og koble den sammen strukturelt til en modul som testes med et simuleringsverktøy. Videre er det et mål å få erfaring med testbenk-kode som kan brukes til å manipulere innganger, og å lese og tolke resultatet av kjøringen fra waveform-vinduet til simuleringsverktøyet. Til sist er det et mål å øke forståelsen for oppbygningen av digital elektronikk, slik som addisjonskretser, slik at det blir lettere å analysere hvordan de virker.

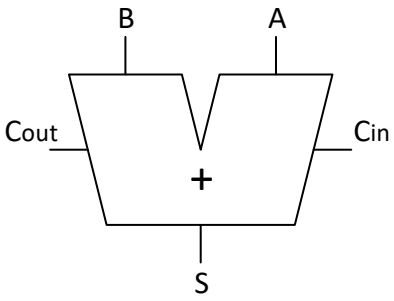
## Oppgave 1, Fulladder og testbenk.

**MERK**: *Det anbefales å ha gjort oblig 1 og innledende ukesoppgaver i kapittel 4 (4.1, 4.3, 4,5 og 4.6) før du går løs på denne oppgaven.*

Skriv en VHDL modul for en fulladder som vist på Figur 1, nedenfor. Modulen skal hete **fulladder.vhd**, og entiteten skal hete også hete fulladder. Bruk navnene a, b og cin for inputene, og s og cout for outputene. Alle inputs og outputs skal være av typen **`std_logic`**.




*Figur 1: Porter i en fulladder*




*Figur 2: Fulladder symbol*

i) Kompiler koden, rett eventuelle feil i kompileringen.

Sammenhengen mellom input og output for fulladderen skal være som angitt i Figur 3:

| a | b | cin | cout | s |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

*Figur 3: Sannhetsverdi for fulladder*

For at testingen av disse verdiene skal gå smidig har vi laget en testbenk for fulladderen som skal brukes for å teste at vi får riktig output i waveform vinduet.

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

entity tb_fulladder is   -- testbenkentiteter er normalt tomme.
end entity tb_fulladder;

architecture behavioral of tb_fulladder is
  -- en komponent er en entitet definert i en annen fil, og som vi vil bruke.
  -- komponentdeklarasjonen må matche entiteten.
  component fulladder is
    port(
      a, b  : in  std_logic;
      cin   : in  std_logic;
      s     : out std_logic;
      cout  : out std_logic
    );
  end component;

  -- Tilordning av startverdi ved deklarasjon gjøres med :=
  signal     tb_a, tb_b, tb_cin : std_logic := '0';

  -- outputs bør ikke få en startverdi i testbenken, da det kan maskere feil.
  signal tb_s, tb_cout : std_logic;

begin
  -- instansiering:
  DUT: fulladder    -- Merkelappen DUT betyr «device under test» som er en av mange
  port map(         -- vanlige betegnelser på simuleringsobjektet.
    a    => tb_a,   -- Mappinger gjøres med =>, til forskjell fra tilordninger som
    b    => tb_b,   -- bruker <= eller :=
    cin  => tb_cin, -- Mappinger kan ses på en ren sammenkobling av ledninger
    s    => tb_s,   -- Vi mapper alltid testenhetens porter til testbenkens signaler
    cout => tb_cout -- Siste informasjon før parantes-slutt har ikke ',' eller ';'
  );

  -- I testbenker kan vi ha prosesser uten sensitivitetsliste..
  -- i slike prosesser kan vi angi tid med «wait» statements, og
  -- vi kan sette signaler flere ganger etter hverandre uten å gi konflikter.
  -- NB: Prosessen vil trigges om og om igjen om vi ikke hindrer det.
  process
  begin
    wait for 10 ns;
    tb_a <= '1';
    tb_b <= '0';
    tb_cin <= '0';
    wait for 10 ns;

    report("Ferdig!") severity note;
    std.env.stop;   -- stopper simuleringen
  end process;
end architecture behavioral;
```

*Figur 4: Uferdig testbenk for fulladderen.*

ii) Lag en fil **tb_fulladder.vhd**, der du skriver inn koden fra Figur 4.
iii) Modifiser koden slik at du får testet alle mulighetene angitt i Figur 3, med 10 ns mellomrom.
iv) Kompilér og rett opp eventuelle feil
v) Simuler med testbenken.
   a. For å simulere med en testbenk, så er det viktig at modulen du skal teste er kompilert før du starter simuleringen.
   b. Start simuleringen med testbenken Simulate->Start simulation->Work->tb_fulladder[1].
   c. I sim-fanen, ekspander tb_fulladder, og velg DUT
   d. Legg til alle signalene til fulladderen til waveformen.
   e. I transcript vinduet skriv: *run –all*

Når simuleringen stopper vil modelsim automatisk hoppe inn i koden der den stoppet. Vanligvis kommer dette vinduet foran wave-vinduet. For å få frem wave-vinduet, kan man enten lukke vinduet med koden, eller trykke på wave-tab'en som dukket opp i underkant av vinduet

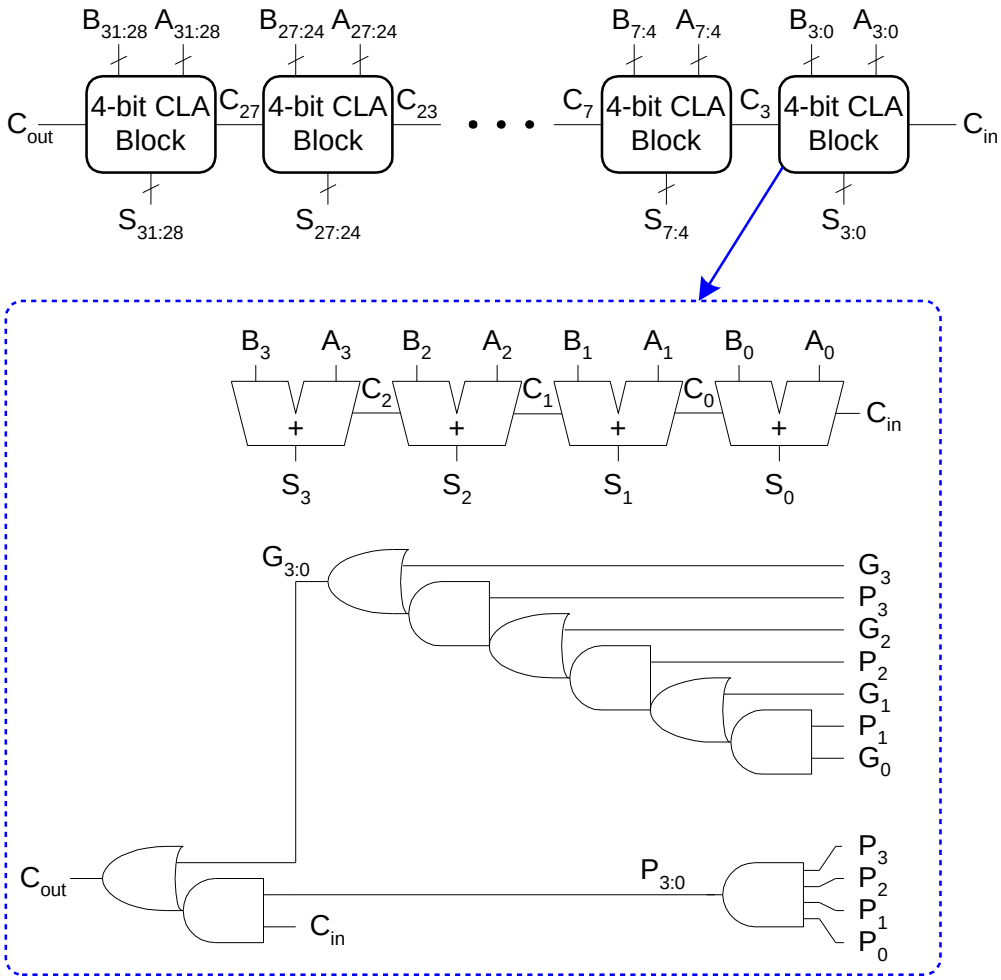
vi) Zoom ut fullt, og kontroller resultatet.
vii) Eksporter waveformen til en png-fil med navn **fulladdertest.png** og en vcd-fil med navn **fulladdertest.vcd.** Alle signalene må være valgt ved eksport av vcd.

*Innlevering i oppgave 1 skal bestå av filene:* **fulladder.vhd, tb_fulladder.vhd, fulladertest.png** *og* **fulladdertest.vcd**

[1] En tidligere bug i Questa (2019.4 eller tidligere) gjorde at testbenkkode ble kuttet vekk dersom optimalisering stod på. Skulle det skje på ny, kan man forsøke å skru den av igjen, men det kan resultere i feilmeldinger ved nyere versjoner.

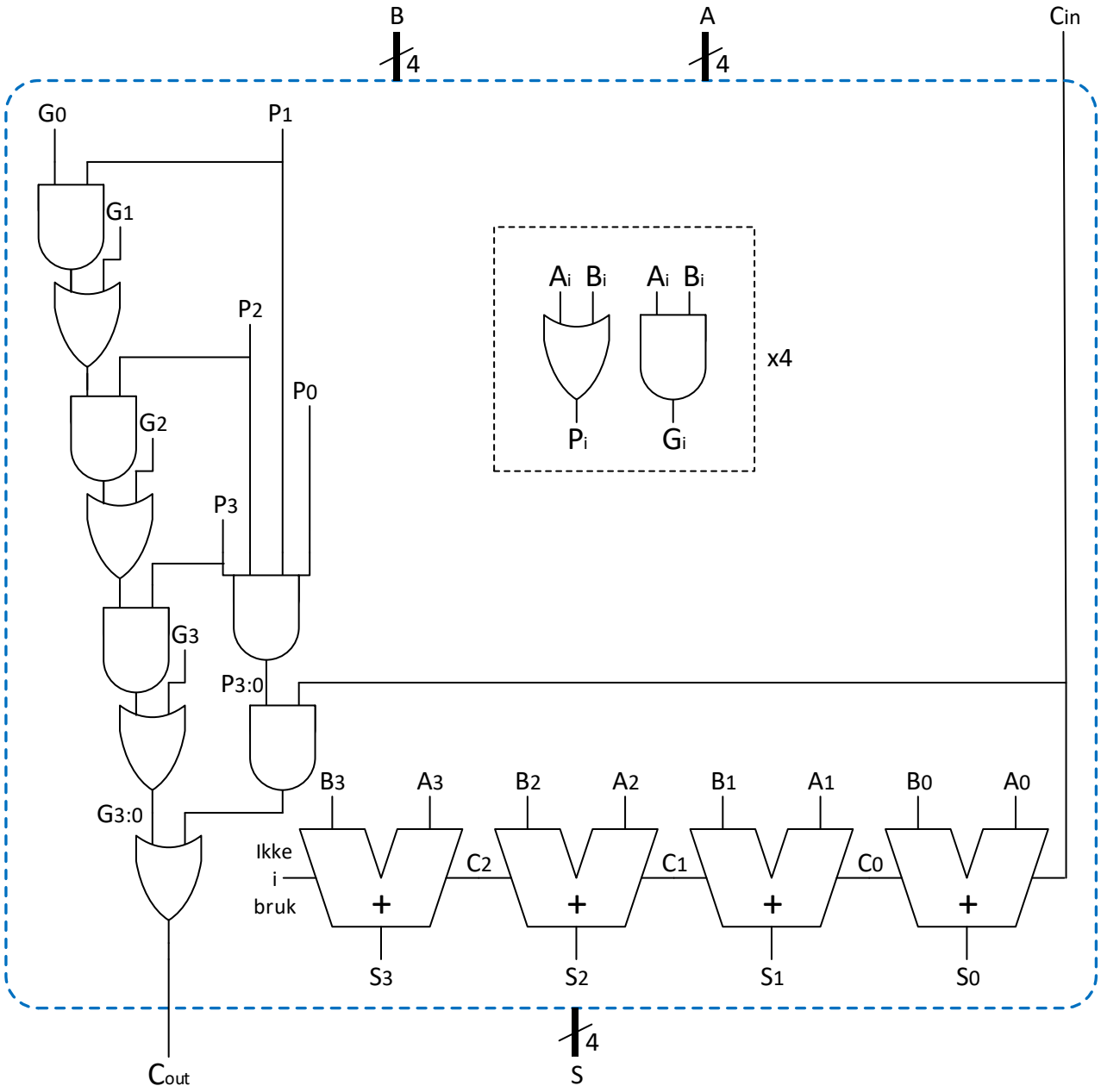
## Carry-lookahead adder (CLA)

I læreboka i kapittel 5.2 er prinsippet til CLA addere beskrevet (s241). I denne oppgaven skal vi lage en 32-bits CLAadder og teste den med simulering. Det er ikke nødvendig å forstå hvorfor CLAadderen vil virke for å fullføre denne oppgaven, men vi håper oppgaven kan hjelpe til å avmystifisere hvordan den er bygget opp.




*Figur 5: 32-bit CLA adder, (Figur 5.6 i læreboka "Digital design and computer architecture").*

Innholdet i CLA blokken er detaljert i Figur 6 .




*Figur 6: Innhold i CLA-blokken*

Utgangspunktet for designet er figur 5.6 på side 242 i læreboka. For å løse oppgaven vil vi dele oppgaven i 3 forskjellige designmoduler. Den innerste modulen har du allerede laget, og er fulladderen fra oppgave 2). Videre har vi en modul for CLA-blokken. CLA blokken skal danne logikken beskrevet i den blå stiplede firkanten i Figur 6. CLA blokken vil benytte fulladderen, og vil derfor ha både strukturell- og dataflytkode. Etterpå setter vi sammen flere CLA blokker i en modul som skal hete CLA_top. Hver designmodul skal ha sin egen testbenk.

```
entity CLA_block is
  port(
    a, b  : in  std_logic_vector(3 downto 0);
    cin   : in  std_logic;
    s     : out std_logic_vector(3 downto 0);
    cout  : out std_logic
  );
end entity CLA_block;
```

*Figur 7: entiteten til CLA-blokk-modulen*

## Oppgave 2, CLA-blokk med testbenk

**MERK**: *Det anbefales å ha gjort ukesoppgaver 4.8, 4.9 og 4.10 før du går løs på denne oppgaven.*

a) Bruk entitetsbeskrivelsen i Figur 7, og lag og kompiler CLA-blokk-modulen

1. Arkitekturen skal hete «mixed» siden du skal benytte en kombinasjon av strukturell- og dataflyt-kode.
2. Inkluder fulladdermodulen som en komponent.
3. Lag egne signaler for «propagate» (p) og «generate» (g). Disse signalene skal være av typen `std_logic_vector`, og skal ha samme antall bit som a og b.
4. Tilordne verdier til «p» og «g» slik at vi får[2] p(i) = a(i) OR b(i), og g(i) = a(i) AND b(i).
5. Lag et carry-signal «c» av typen `std_logic_vector`, la antall bit være 5 (fra 4 ned til 0).
6. Lag et eget signal «p30» og «g30» for å beregne $P_{3:0}$og $G_{3:0}$ i henhold til Figur 5.
7. Bruk p30 og g30 til å beregne cout.
    i. Hint: bruk en reduksjonsoperator for p30
    ii. Hint: bruk paranteser for å holde orden på g30
8. Sett c(0) til cin.
9. Opprett (instansier) de fire fulladderne og koble dem med port maps til signalene a, b, c. Velg én av de to metodene å gjøre det på:

    ***Enten:*** *Gi hver av de fire fulladderne en egen merkelapp og instansier dem, og koble til portene med port-mapping*. (Som instansieringen av fulladderen i tb_fulladder).

    **Eller** (vanskelig): I VHDL kan man bruke løkker med **`for + loop`**, `eller` **`for + generate.`** Bruk av løkker gir kompakte løsninger som skalerer bedre enn om man skriver ut løsningen manuelt. Når man oppretter komponenter brukes **`generate`** *(ved bruk av eksisterende komponenter og signaler brukes* **`loop`***.)*
    *Løs oppgaven ved å benytte* ***`for …+ generate`****.*

    Eksempel:

```
min_løkke: for i in 1 to 5 generate ny_komponent: min_komponent
    port map(
      epler => eplesignal(i),
      pærer => pæresignal(i)
    );
    end generate;
```

*Figur 8: for-generate-løkke*

*Ved behov, søk etter mer detaljert beskrivelser og flere eksempler på internett.*

[2] Dette tilsvarer det læreboka sier i kap 5.2.1, under «Carry-Lookahead Adder». Noen kilder (som wikipedia eller geeksforgeeks) vil fremheve at Propagate lages med XOR, og ikke OR slik læreboka gjør. Hvis man ønsker en streng tolkning av propagate, der man ikke propagerer når man genererer mente, så blir XOR eneste riktige løsning. I praksis kan begge metoder benyttes, siden det er likegyldig om vi propagerer carry når vi lager den. Spesielt interesserte vil også kunne merke seg at avhengig av teknologien som benyttes (typisk CMOS), vil noen port-typer kan lages med raskere, færre, eller mindre transistorer; men dette er utenfor pensum i IN2060.

b) Lag en testbenk for å teste CLA_blokken. Testbenken skal hete **tb_CLA_block.vhd**. Benytt `assert` slik at testbenken stopper ved feil.

Eksempel på **assert**-statement:

```
assert(i = j) report ("i er ulik j") severity failure;
```

**assert** rapporterer når det boolske uttrykket inni parantesen er usant.
Alvorlighetsgraden, **severity,** har fire nivåer: *note, warning, error, failure*.
Bare *failure* stopper simuleringen. Report kan også benyttes alene for å alltid gi beskjed.

*Figur 9: assert-statement*

Velg én av metodene under:

**Enten**: Lag den nye testbenken ved å kopiere og modifisere testbenken i 1b), slik at den passer for CLA blokken. Velg ut minst 6 tallpar som du legger sammen med CLA blokken. Sørg for at noen vil lage mente (carry)- og noen ikke gjør det. Bruk `assert` statement til å teste om resultatene fra CLA-blokken er riktig. Bruk «`severity failure`» ved feil. Etter alle testene er gjennomført skal testbenken gi en tekstbeskjed om testen foregikk feilfritt.

**Eller** (Vanskelig): Test at CLA modulen oppfører seg slik den skal ved å lage en testbenk som går igjennom alle muligheter for a, b og cin.

i. Bruk `for …+ loop` for å lage alle verdiene for a, b og cin.
ii. Bruk `assert` for å sjekke at verdiene stemmer, bruk «`severity failure`» ved feil.
iii. Bruk `report` for å rapportere når testbenken er ferdig.

**Hint (for begge alternativene):**

i. CLA blokken tar inn to fire bits signaler (a og b), samt carry (cin). Testbenken må sette disse signalene og sjekke verdiene på summen (s) og carry out (cout).
ii. For at simuleringen skal vise forskjellene, må du vente mellom hver gang du setter og tester noe. Du bør vente noenlunde like lenge hver gang, slik at du kan sjekke resultatet på waveformen.

*Innlevering i oppgave 2 skal bestå av* VHDL filen til CLA blokken fra a), testbenken fra b) *og* en tekstfil med kopi av outputen i konsollvinduet for siste kjøring.

## Oppgave 3, CLA-toppmodul med strukturell kode og testbenk

```
entity CLA_top is
  generic(
    width : positive := 32;
  );
  port(
    a, b  : in  std_logic_vector(width-1 downto 0);
    cin   : in  std_logic;
    sum   : out std_logic_vector(width-1 downto 0);
    cout  : out std_logic
  );
end entity CLA_top;
```

*Figur 10: entiteten til top-modulen til CLA adderen.*

a) Bruk entitetsbeskrivelsen i Figur 10 og lag og kompiler toppmodulen slik at den kan utføre 32 bits addisjon. CLA_top skal benytte åtte CLA-blokker til å utføre beregningen. Du velger selv hvordan du instansierer komponentene.

b) Lag en testbenk som tester ut 10 forskjellige tallkombinasjoner etter tur og rapporterer eventuelle feil. Tallkombinasjonen bør inneholde noen tall som bruker, og noen som ikke bruker de mest signifikante bitene i a og b. Eksporter waveformen som en vcd fil **CLA_top.vcd** som viser kjøringen med alle input og output.

*Innleveringen i oppgave 3 skal bestå av toppmodulen* **CLA_top.vhd**, *testbenken* **tb_CLA_top.vhd** *og* en waveform **CLA_top.vcd** *som viser kjøringen med alle verdiene for input og output*.

**Hint (valgfritt å benytte):**

I VHDL kan man ikke uten videre oversette mellom tall (integer) til andre typer, slik som std_logic, men det finnes biblioteker som har funksjoner som kan gjøre det.

For tallkonverteringer benytter vi biblioteket `IEEE.numeric_std`

Her er et eksempel på deklarering av biblioteket

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

Hvis vi skal konvertere et heltall til en **std_logic_vector**, må vi først bestemme oss om vi vil ha med fortegnsbit eller ikke. For å benytte fortegn har biblioteket en type som heter **signed**, mens uten fortegn har vi **unsigned**. Disse typene består av std_logic_vectors, og man må angi bitbredden på samme vis. Når vi konverterer et tall til f.eks unsigned, så må funksjonen vi bruker ha beskjed om hvor mange bit vi skal ha. Her følger eksempel på konvertering fra **integer** til **std_logic_vector** og fra **std_logic_vector** til **integer** uten fortegn:

```
signal my_sig : std_logic_vector(31 downto 0);
signal my_int : integer;

…

my_sig <= std_logic_vector(to_unsigned(my_int,32));

my_int <= to_integer(unsigned(my_sig));
```