INF3430 Høsten 2009

Designflyt
Utviklingsverktøyene Modelsim og Xilinx ISE

1 Designflyt og verktøy 4

1.1 Designflyt for programmerbar logikk 5
1.1.1.1 Design entry 5
1.1.1.2 Register Transfer Level (RTL)-simulering 6
1.1.1.3 Syntese 6
1.1.1.4 Place and Route(Fitting) og timing simulering 6
1.1.1.5 Statisk timing analyse 6
1.1.1.6 Device Programming 6

1.2 Verktøy benyttet i kurset 7

2 Hvordan komme i gang med Modelsim 8

2.1 Oppretting av prosjekt 8

2.2 Forberedelse til VHDL-simulering med Modelsim 8

2.3 Eksempelfiler 8
2.3.1 Designfilen first.vhd 8
2.3.2 Testbenken tb_first.vhd 9
2.3.2.1 VHDL-Testbenk 9

2.4 Oppretting av prosjekt i Modelsim 10

2.5 Kompilering av kildefiler 11
2.5.1 Litt om Working library 11
2.5.2 Kompilering av VHDL kildefiler 12

2.6 Simulering 13

2.7 Makrofiler (do-filer) 15

3 Hvordan komme i gang med Xilinx ISE 16

3.1 Oppretting av prosjekt 16

3.2 Bruk av ISE 19

3.3 Tilordning av pinner (Pin number constraints) 20

3.4 Timing constraints 23

3.5 Syntesen 23

3.6 Design implementation 24
3.6.1 Translate 24
3.6.2 Map 24
3.6.3 Place & Route 24

3.7 Videre simuleringer 25
3.7.1 Automatisert oppstart av Modelsim 26
3.7.2 Manuell oppstart av Modelsim 27

3.8 Device Programming 29
3.8.1 Tilkoblinger 29

3.8.2 Nedlasting av konfigurasjon direkte til FPGA via JTAG 29
3.8.3 Programmering av konfigurasjonsminnet via JTAG 34

4 Editering av VHDL-kode 39

4.1 Valg av editor 39

4.2 Indentering 39
4.2.1 Eksempel på uindentert og bra indentert kode: 39
4.2.2 Tab/space for indentering 39

4.3 Kommentarer i koden og variabelnavn 40

4.4 Bruk av tastatur 40
4.4.1 Taster og kombinasjoner en må kunne 40
4.4.2 Spesielle triks for Notepad++ 40

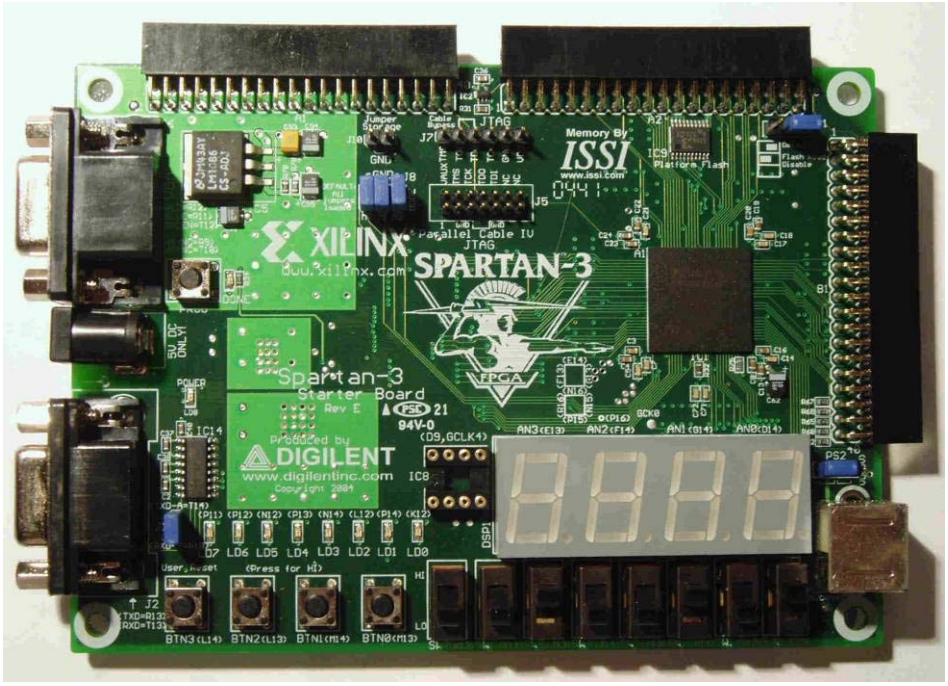
1 Designflyt og verktøy

I dette kurset skal vi bli kjent med to kraftige programmer for FPGA design:

1. Modelsim. VHDL/Verilog/System C/System Verilog simulator
2. Xilinx ISE (Integrated Synthesis Environment). Syntese og Place & Route/Device fitting verktøy

Dette notatet er skrevet som en "kokebok" for hvordan man kommer igjennom designprosessen fra A til Å første gangen, og uten for store sidesprang i bruk av programmene. Kokeboken forutsetter at man har tilgang til Spartan-3 starter kit board. Dokumentasjon for dette kan man hente fra:
http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf
Papirkopi av denne finnes også utlagt på laben.

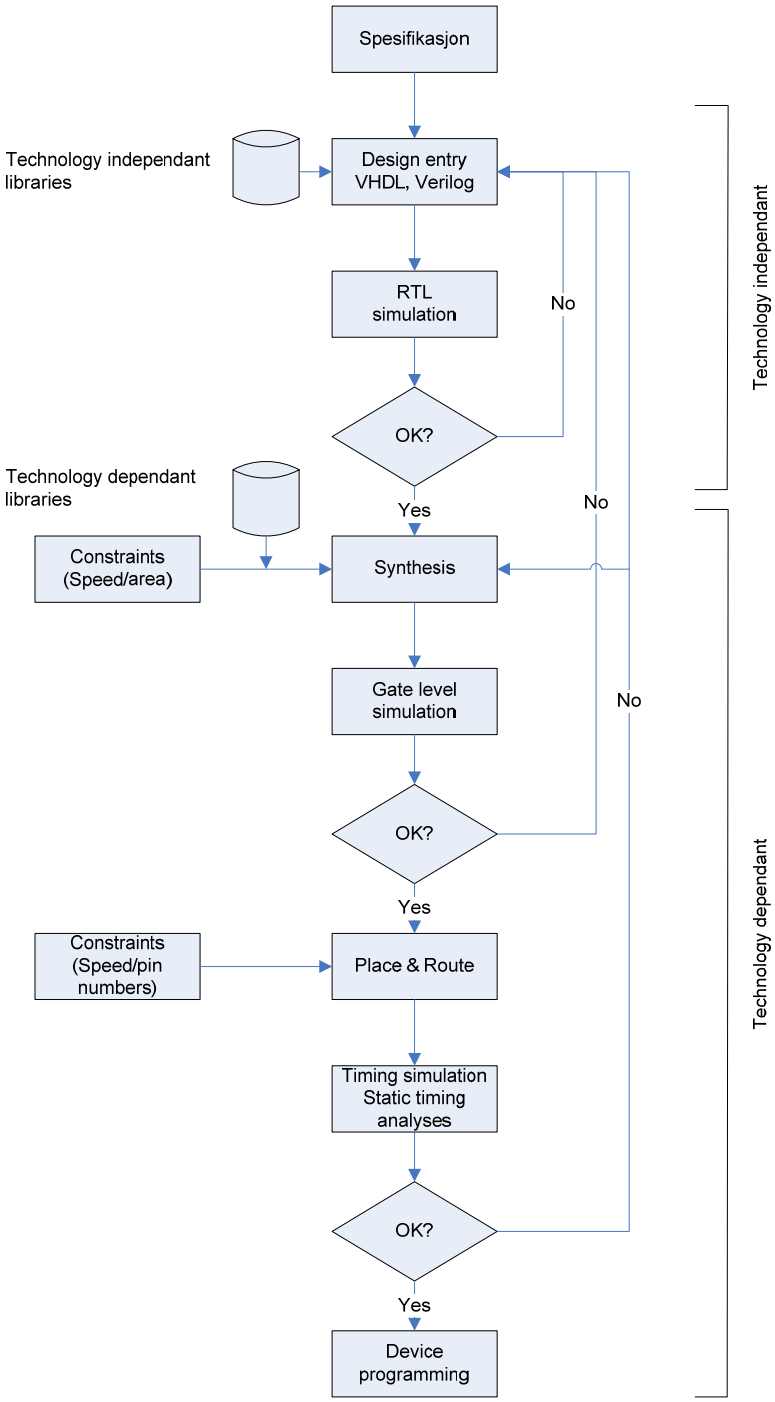
Hvis man vil ha en mer inngående tutorial henvises til:
http://www.xilinx.com/support/techsup/tutorials/tutorials10.htm



Figur 1. Spartan-3 starter board

1.1 Designflyt for programmerbar logikk

Figuren nedenfor viser en generell designflyt basert på et HDL (Hardware Description Language) for en FPGA.



Figur 2. Designflyt for FPGA

1.1.1.1 Design entry

Man kan tenke seg flere måter å beskrive et design på. Den tradisjonelle måten er å tegne et koblingsskjema. Dette er en metode som man har gått bort i fra når det gjelder digitale design fordi det er vanskelig å vedlikeholde. Dessuten er det en formidabel jobb å tegne og ikke minst vedlikeholde skjema for et design som skal kanskje inneholde noen hundre tusen porter og flip-flop'er.

Rundt midten av 1980-tallet ble det utviklet egne programmeringsspråk til å beskrive hardware, såkalte HDL (Hardware Description Language). VHDL ble standardisert i 1987. Det er mye enklere å vedlikeholde og utvide design som foreligger som tekst enn et grafisk design.
I kurset vårt skal vi benytte VHDL som design entry.

1.1.1.2 Register Transfer Level (RTL)-simulering

Den første simuleringen i Figur 2 er en ren funksjonell simulering, av f.eks. VHDL-kode. Til dette må vi benytte en VHDL-simulator. Dette abstraksjonsnivået kalles RTL-nivå. Ideelt sett trenger vi ikke ta stilling til hvilken teknologi designet skal inn i på dette tidspunktet. Men ofte er det slik at vi må endre koden slik at den tar hensyn til restriksjoner/begrensninger i den teknologien vi til slutt velger. Det kan være at vi må ta hensyn til klokkedistrubusjon, globale reset, styring av tristateutganger osv. Etter at simuleringen er ferdig og vi er fornøyd går vi over til syntesen (se 1.1.1.3). Men før det må man gjøre et teknologivalg (type krets/kretsfamilie).
Mesteparten av simulerings- og debuggingsjobben gjøres på RTL-nivå fordi der har vi meget god observerbarhet.

1.1.1.3 Syntese

Syntesen er en prosess som overfører en RTL-beskrivelse over til en portnivå (gatelevel) beskrivelse.
I en RTL-beskrivelse har man som regel at alle registre (flip-flop'er) og klokker er synlige. Gatelevel vil si sammenkopling av basiskomponenter (AND, OR, NOT osv) i et teknologiavhengig bibliotek.
Vi påvirker resultatet av syntesen ved å lage såkalte "constraints". Dette er krav som vi setter for å hjelpe verktøyet i en ønsket retning. Typiske syntese "constraints" kan være hastighet kontra plass "constraints". Andre "constraints" kan være enkoding av tilstandsvektorer i FSM'er (Finite State Machine).
Resultatet av syntesen kan også være boolske ligninger dersom designet skal inn i en CPLD.
Det er som regel mulig å gjøre en simulering etter syntesen. Dette vil være en verifisering av at syntesen har gitt rett resultat.

1.1.1.4 Place and Route(Fitting) og timing simulering

Dersom simuleringen etter syntesen er korrekt kan man gå over til prosessen der man plasserer det syntetiserte designet inn i den teknologien man har valgt. Denne prosessen kalles "Place and Route" for FPGA-er og "Fitting" for CPLD-er. Verktøyene man benytter i denne prosessen lager kretsleverandørene. Tilordning av pinner og krav til ytelse (klokkefrekvens) er viktige "constraints" i denne delen av designprosessen.
Etter at designet har gått igjennom en "Place and Route" eller "Fitting" prosess, vil man gjerne verifisere både funksjonalitet og timing på designet. Og man har mulighet til å generere simuleringsfiler på diverse format som kan benyttes til dette. Er denne simuleringen i orden, er man så og si ferdig.

1.1.1.5 Statisk timing analyse

Det kan være ganske tidkrevende å kjøre en timingsimulering av et større design. Et alternativ som benyttes mye er å gjøre en statisk timinganalyse. Det finnes egne verktøy for dette. Et vanlig krav vi har til timing er max. klokkefrekvens. Det som begrenser denne er forsinkelser i kombinatorisk logikk mellom registre. Andre krav kan være "clock to output delay", input "set-up" og "hold" tider. Statisk timing analyseverktøy identifiserer "worst case" baner (path'er) slik at vi kan få en oversikt over ytelsen til kretsen. Korrekt funksjon kan verifiseres ut fra en ren funksjonell simulering på det "routede" designet.

1.1.1.6 Device Programming

Da gjenstår bare det vi har kalt "device programming", dvs. å få designet inn i den ønskede kretsen. Dette kan gjøres på svært mange forskjellige måter. F.eks. krever SRAM-baserte FPGA-er at man benytter en ekstern PROM/EPROM/Flash EPROM eller lignende for å lagre kretskonfigurasjonen (designet). Anti-fuse baserte FPGA-er brennes en gang, og det er ingen mulighet for endringer.
I en debuggingsfase er det vanlig å laste programmeringsfilen rett ned i kretsen fra en PC via en programmeringskabel.

I en CPLD lagres konfigurasjonen internt i kretsen, og det finnes varianter som er EPROM basert som programmeres elektrisk og som kan slettes med UV-lys. Det finnes EEPROM baserte som både kan programmeres og slettes elektrisk. De mest interessante variantene er de som kan programmeres og reprogrammeres mens kretsen sitter loddes fast på det ferdige kortet. Disse kalles ISR CPLD'er (In System Reprogrammable).

1.2 Verktøy benyttet i kurset

Siden vi skal benytte en FPGA fra Xilinx med tilhørende verktøy, blir designflyten for denne litt annerledes en den generelle. For Xilinx og andre teknologier kan man også benytte generelle syntese verktøy. Eksempel på slike verktøy er Precision fra Mentor Graphics og Synplify fra Synplicity. Felles for disse er at de kan syntetisere til mange forskjellige målteknologier også ASICs (Application Specific Integrated Circuit). I kurset skal vi benytte syntese og place og routeverktøy som er spesifikke for Xilinx. Simulatoren Modelsim er imidlertid helt generell og uavhengig av teknologi.
På laboratoriet vil det være installert fullversjoner av Xilinx ISE og ModelSim SE 6.4 (vi har 15 lisenser for ModelSim SE). Det kan imidlertid lastes ned gratisversjoner av både Xilinx ISE (Webpack) og en versjon av Modelsim, Modelsim XE III Starter (Xilinx Edition). Dette er en fullt ut funksjonell versjon, men ytelsen går kraftig ned når antall kodelinjer overstiger 10000. Den kan lastes ned fra: http://www.xilinx.com/webpack/classics/wpclassic/index.htm. I kurset benytter vi versjon 10.3.03 og vi anbefaler at dere laster ned samme versjon. Merk at dere må opprette en brukerkonto hos Xilinx før nedlastning kan starte.

2 Hvordan komme i gang med Modelsim

Modelsim kan benyttes alene eller benyttes integrert i Xilinx ISE. Vi skal først se på hvordan man benytter Modelsim alene. Gjennomgangen nedenfor gjelder både for Modelsim XE starter edition og Modelsim SE ver. 6.2e (og oppover).

2.1 Oppretting av prosjekt

Prosjekthåndtering er viktig i alle DAK-system. Meningen med et prosjekt er å skape en ordnet lagerstruktur for alle filer tilhørende et design.

2.2 Forberedelse til VHDL-simulering med Modelsim

Beskrivelsen av Modelsim tar utgangspunkt i et enkelt VHDL eksempel, der vi modellerer en 4-bits teller. Telleren resettes asynkront til 0 av RESET signalet og trigger på positiv flanke av klokken CLK. Videre kan telleren resettes synkront til tallverdien på inngangen INP ved å gi en positiv puls på signalet LOAD. Når telleren når 15 (F hex), går signalet MAX_COUNT aktivt.

Vi skal først se på hvordan vi kan foreta en funksjonell VHDL-simulering av telleren. Dette eksemplet er ment som en første introduksjon til Modelsim og viser langt fra alle muligheter man har der, men én farbar vei. Når vi har gjort den første simuleringen skal vi fortsette med å ta den samme koden inn i Xilinx ISE og syntetisere og plassere designet inn i en Spartan 3 FPGA.
Etter dette skal vi kjøre en timing simulering, og når denne har verifisert at kretsen virker som den skal, programmerer vi kretsen.

2.3 Eksempelfiler

2.3.1 Designfilen first.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity FIRST is
  port
  (
    CLK        : in  std_logic; -- Klokke fra bryter CLK1/INP1
    RESET      : in  std_logic; -- Global Asynkron Reset
    LOAD       : in  std_logic; -- Synkron reset
    INP        : in  std_logic_vector(3 downto 0); -- Startverdi
    COUNT      : out std_logic_vector(3 downto 0); -- Telleverdi
    MAX_COUNT  : out std_logic  -- Viser telleverdi
  );
end FIRST;

--  Arkitekturen under beskriver en 4-bits opp-teller.  Når telleren når
--  maksimal verdi går signalet MAX_COUNT aktivt.

architecture MY_FIRST_ARCH of FIRST is

  --  Område for deklarasjoner
  signal COUNT_I : unsigned(3 downto 0);

begin
  --  Her starter beskrivelsen

  COUNTER :
  process (RESET,CLK)
  begin
    if(RESET  = '1') then
```

```
      COUNT_I <= "0000";
    elsif (CLK'event and CLK = '1') then
      -- Synkron reset
      if LOAD = '1' then
        COUNT_I <= INP;
      else
        COUNT_I <= COUNT_I + 1;
      end if;
    end if;
  end process COUNTER;

  -- Concurrent signal assignments
  COUNT <= std_logic_vector(COUNT_I);
  MAX_COUNT <= '1' when COUNT_I = "1111" else '0';

end MY_FIRST_ARCH;
```

2.3.2 Testbenken tb_first.vhd

Å simulere et elektronikkdesign går ut på at man påtrykker stimuli på innganger og sjekker at kretsen gir en korrekt respons på utgangene.

2.3.2.1 VHDL-Testbenk

Den vanligste og mest slagkraftige måten å lage stimuli på er å lage en VHDL **Testbenk**. Det betyr at vi lager alle stimuli input i VHDL. (Alternativt kunne man benyttet simuleringskommandoer).
Bruk av VHDL testbenk gir mange fordeler, bla. at simuleringen kan portes over på en annen standard VHDL-simulator, fordi man ikke blir avhengig av simulatorens kommandospråk. Å benytte VHDL testbenker er meget fleksibelt og slagkraftig fordi man har hele VHDL språket til rådighet. Bl.a. kan man inkludere mye av omgivelsene (andre komponenter, evt. et helt kort) i testbenken.
Eksemplet under forutsetter at man har lært en del VHDL, bl.a. om komponent instansisering, men vi velger å ta det med her for å vise en mal for hvordan man kan lage en enkel testbenk. Vi skal komme tilbake senere med mer avansert bruk av testbenker.

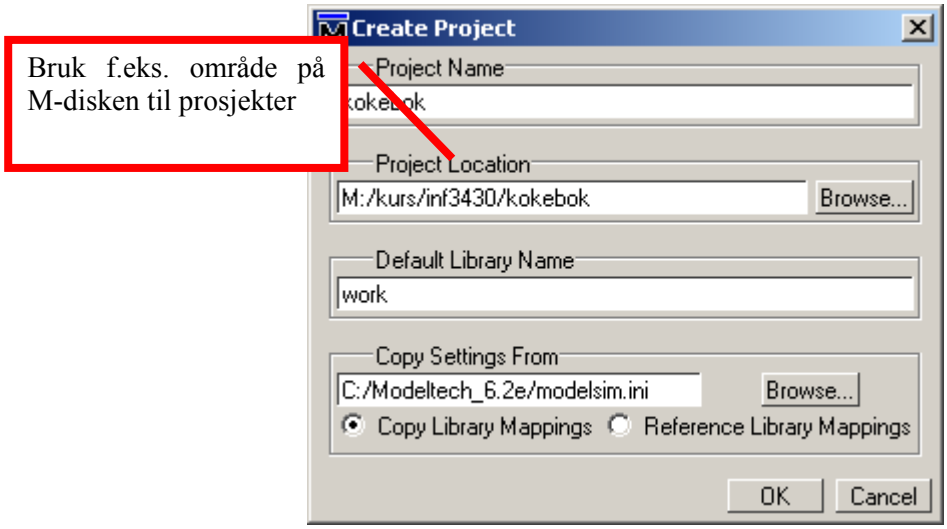
Det er fornuftig å ha en navnekonvensjon på filnavn. F.eks. kan vi kalle testbenk filen for tb_first.vhd, der tb_ betyr at dette er en testbenkfil.

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

Entity TEST_FIRST is
  -- generic parameters
  -- port list
  -- Entiteten for en testbench er gjerne tom
end TEST_FIRST;

architecture TESTBENCH of TEST_FIRST is
  -- Område for deklarasjoner
  -- Komponent deklaration

  Component FIRST
    port
    (
      CLK        : in  std_logic; -- Klokke fra bryter CLK1/INP1
      RESET      : in  std_logic; -- Global Asynkron Reset
      LOAD       : in  std_logic; -- Synkron reset
      INP        : in  std_logic_vector(3 downto 0); -- Startverdi
      COUNT      : out std_logic_vector(3 downto 0); -- Telleverdi
      MAX_COUNT  : out std_logic -- Viser telleverdi
    );
  end Component;

  signal  CLK        : std_logic := '0';
  signal  RESET      : std_logic := '0';
  signal  LOAD       : std_logic := '0';
  signal  INP        : std_logic_vector(3 downto 0) := "0000";
  signal  COUNT      : std_logic_vector(3 downto 0);
  signal  MAX_COUNT  : std_logic;

  constant Half_Period : time := 10 ns;  --50Mhz klokkefrekvens

begin
```

```
  --Concurrent statements

  --Instantierer "Unit Under Test"
  UUT : FIRST
  port map
  (
    CLK        =>  CLK,
    RESET      =>  RESET,
    LOAD       =>  LOAD,
    INP        =>  INP,
    COUNT      =>  COUNT,
    MAX_COUNT =>  MAX_COUNT
  );

  -- Definerer klokken
  CLK <= not CLK after Half_Period;

  STIMULI :
  process
  begin
    RESET <= '1', '0' after 100 ns;
    INP <= "1010" after Half_Period*12;
    wait for 2*Half_Period*16;
    LOAD <= '1', '0' after 2*Half_Period;
    wait;
  end process;

end TESTBENCH;
```

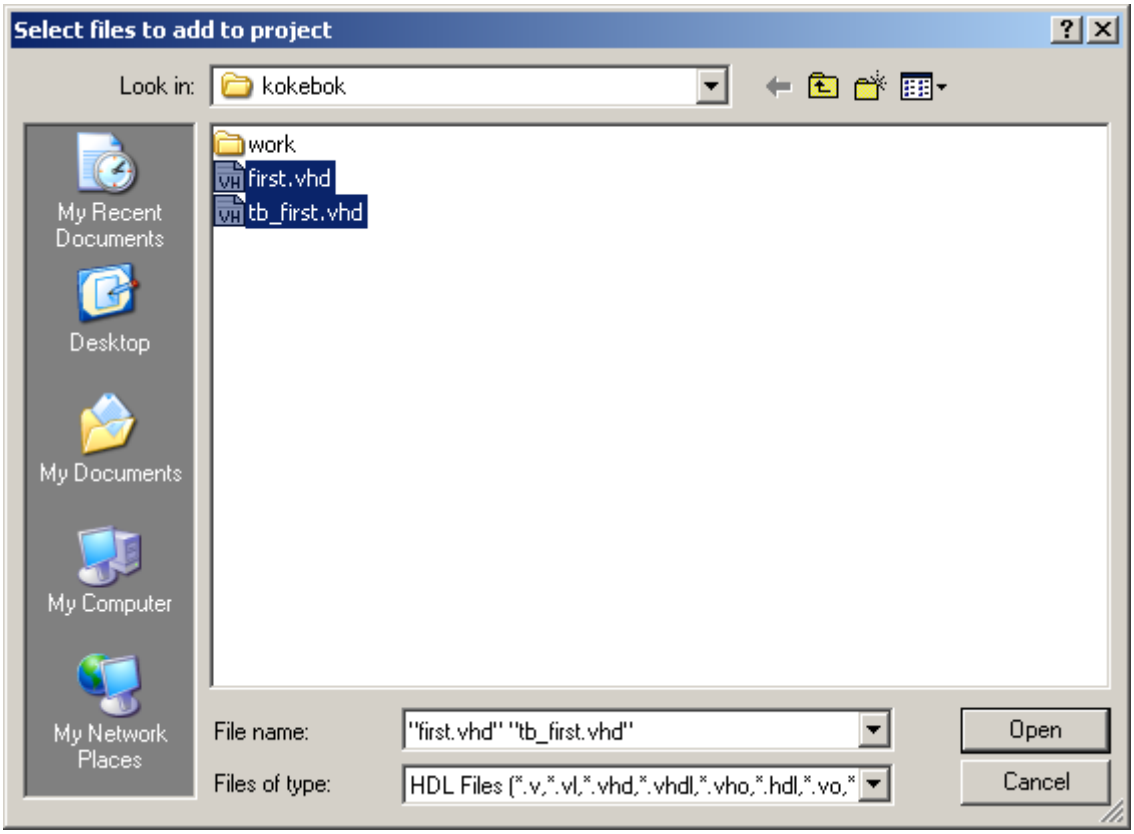
2.4 Oppretting av prosjekt i Modelsim

Man starter modelsim ved å gå inn på **Start⇒Programs⇒Modelsim (XE eller SE) ⇒Modelsim.** Eventuelt kan Modelsim startes fra desktop(skrivebordet).
Velg **File⇒New⇒Project** (etter å vært igjennom en eventuell velkomsthilsen). Her velger dere f.eks. en katalog på deres private nettverksdisk M.



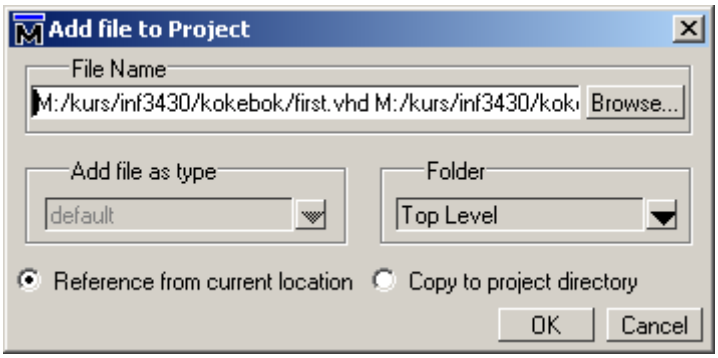
Figur 3. Oppretting av prosjekt (1)

Etter man har opprettet prosjektet får man spørsmål om man vil legge til filer til prosjektet. Dette kan også gjøres når som helst ellers. Oftest er ikke alle kildefiler i prosjektet klare under første simulering, så man får behov for å legge til flere filer senere.
Man søker seg fram til området hvor kildefilene ved å velge ”Add Existing File”, deretter ”Browse” og velger de som skal være med ved ctrl+klikk for hver fil en ønsker lagt til:



Figur 4. Legge til kildefiler(1)

Det anbefales at man ikke tar kopi av kildefilene til prosjektområdet, men at de refereres til der de opprinnelig lå. Det er viktig å ha en velordnet lagerstruktur på kildefiler.



Figur 5. Legge til kildefiler(2)

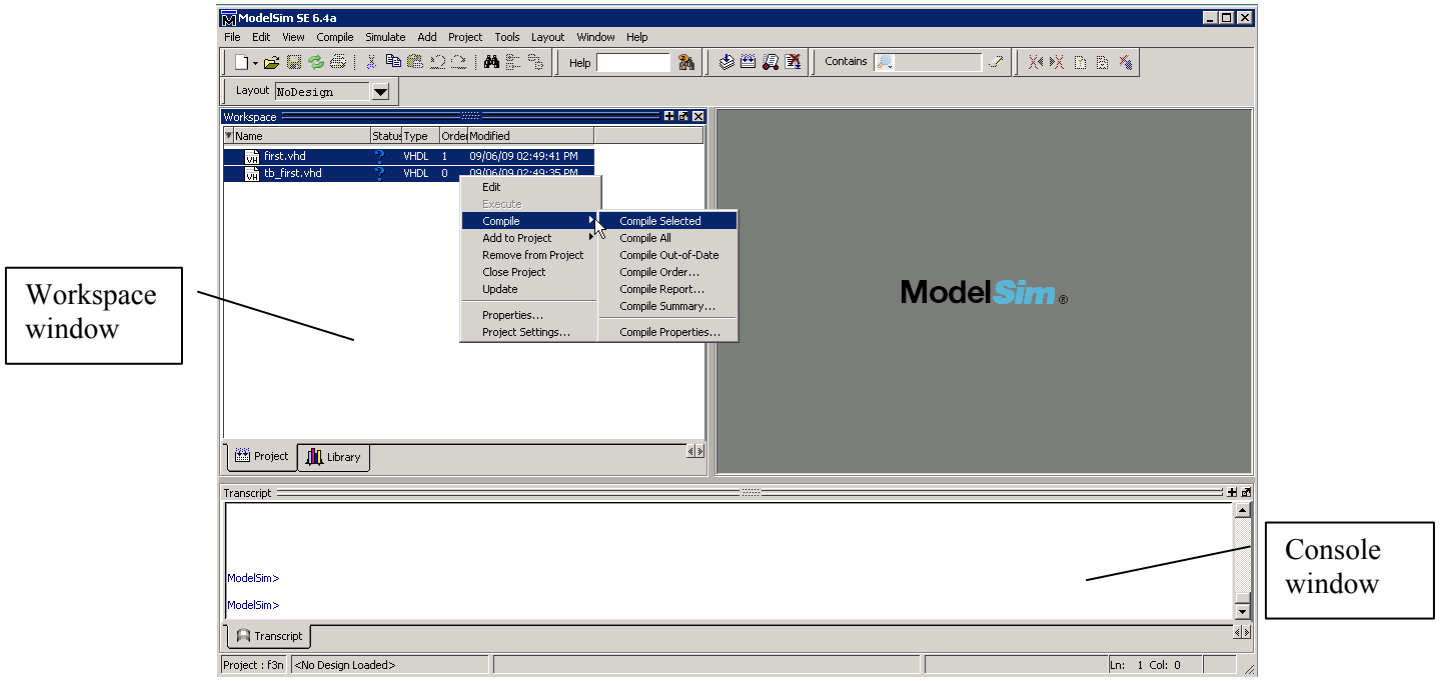
2.5 Kompilering av kildefiler

2.5.1 Litt om Working library

Working library eller work har en spesiell betydning i VHDL. Det refererer til VHDL-filer som er under utvikling og uttesting. Det er meget viktig for en VHDL-simulator og synteseverktøy å vite hvilke bibliotek som til en hver tid er working. Vi kompilerer, simulerer og syntetiserer alltid på modeller som tilhører **work**. Vi kan ha mange brukerbiblioteker, men bare **ett** bibliotek kan være working. Dersom vi i en kildefil refererer til work betyr det at simulatoren forventer å finne aktuell designenhet (mer om designenheter senere i kurset) i biblioteket som vi laster designet inn i simulatoren fra. Omvendt så kalles biblioteket vi laster et design fra for working library. Dette betyr videre at VHDL krever at vi har et aktivt forhold til biblioteker og til hvilke bibliotek designenheter blir kompilert. Dersom vi skal benytte designenheter fra andre bibliotek må vi referere til det logiske navnet på biblioteket. Vi skal se nærmere på bruk av biblioteker senere i kurset.

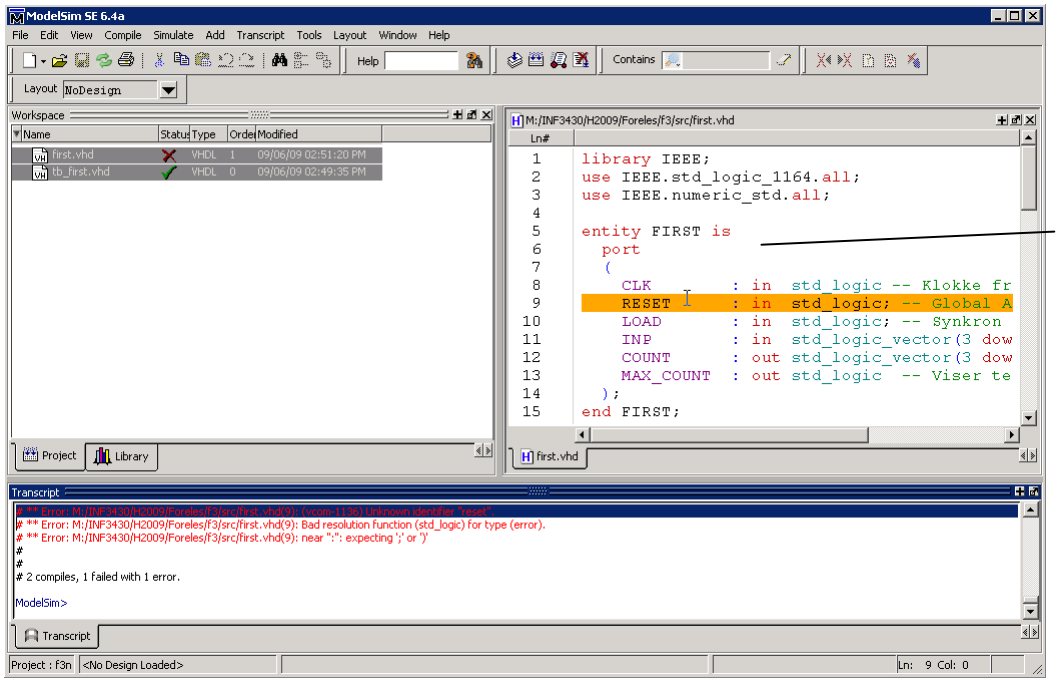
2.5.2 Kompilering av VHDL kildefiler

For å kompilere kan man merke ønskede filer, **høyreklikk⇒Compile Selected.** Dersom man vil kompilere alle filene i prosjektet kan man gjøre dette ved **høyreklikk⇒Compile All**:



Figur 6. Kompilering av VHDL-kildefiler

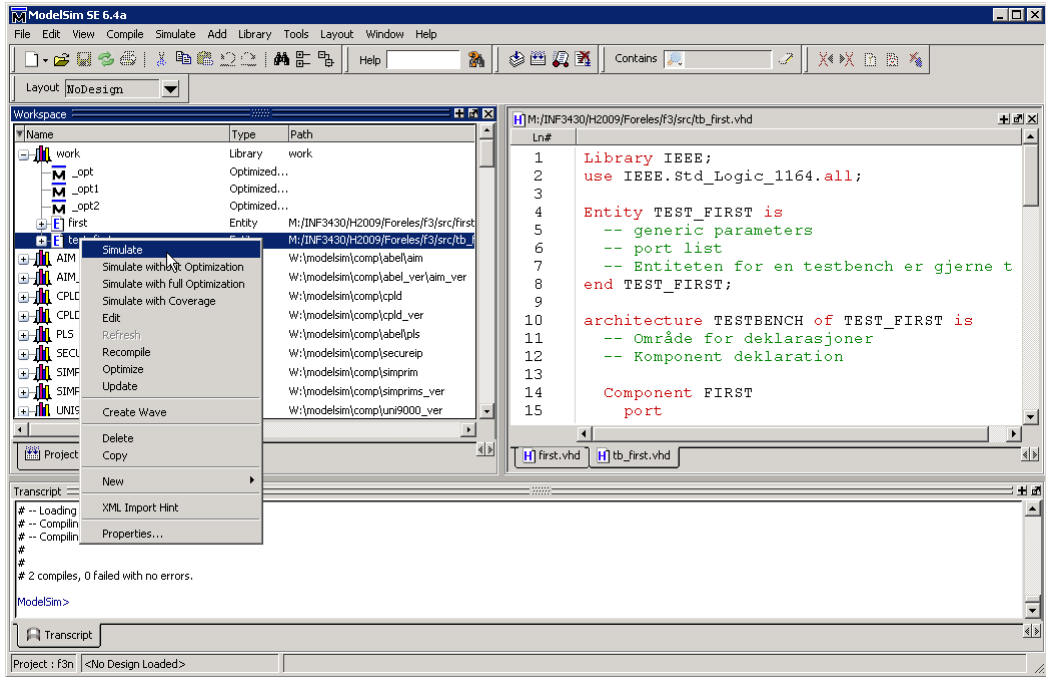
Sannsynligheten for syntaksfeil under første kompilering er bortimot 100 %. I Modelsim har vi som i andre kompilatorer en ”error parser” som viser oss hvilke linjenummer i hvilke filer som har syntaksfeil. Det er en prosjekt opsjon som styrer om man vil vise ”compiler output” i simulatorens ”consol” vindu. Denne opsjonen kan være nyttig å ha slått på (Velg ”Project Window” i ”**Workspace”⇒høyreklikk⇒”Project Setting”⇒Huk av ”Display Compiler output**”). Da kan man bare klikke på feilmeldingen og bli styrt rett til linjen hvor det er feil eller rett i nærheten. Vær oppmerksom på at en liten feil ofte produserer mange følgefeil. I eksemplet i Figur 7 under manglet et semikolon og det førte til 3 andre feilmeldinger.



Figur 7. Error parser

2.6 Simulering

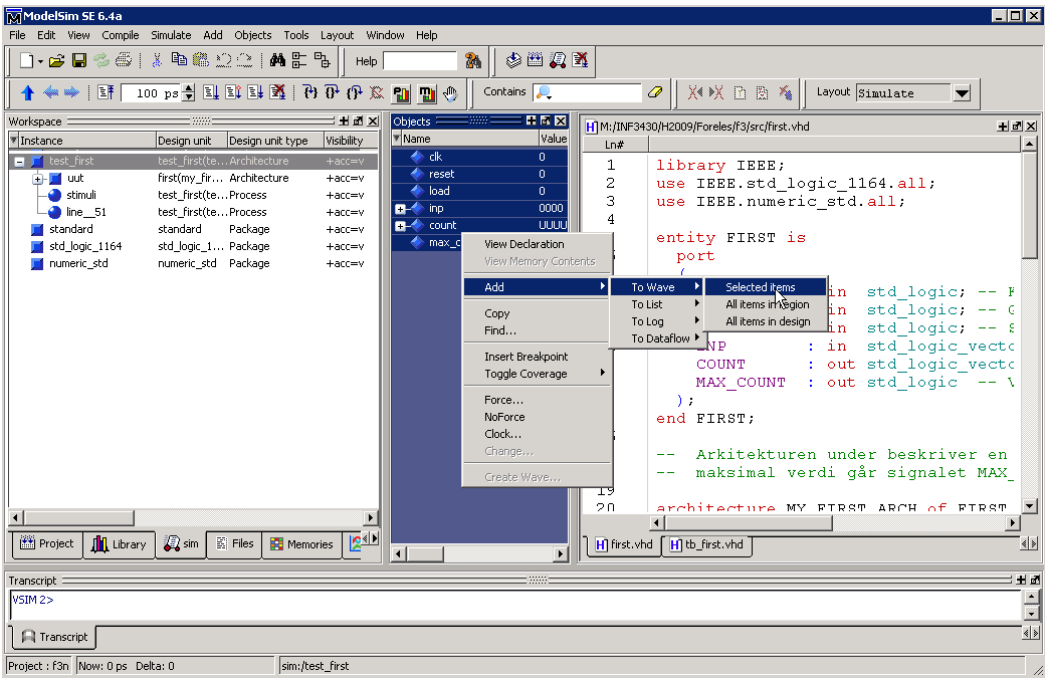
Når kildefilene er kompilert er vi klare til å simulere. Da velger man "Library" i "workspace" og i dette tilfelle går man til biblioteket ved navn **work**, hvor hver designenhet er kompilert.
For å få fram designenheter i de forskjellige bibliotekene kan man ekspandere biblioteket ved å klikke på "+" merket foran biblioteket. Vi velger å laste inn entiteten "test_first" som er entiteten til vår testbenk. Denne representerer det øverste nivået i vårt designhierarki. Merk denne entiteten⇒**høyreklikk⇒Simulate**:



Figur 8. Klargjøring for simulering

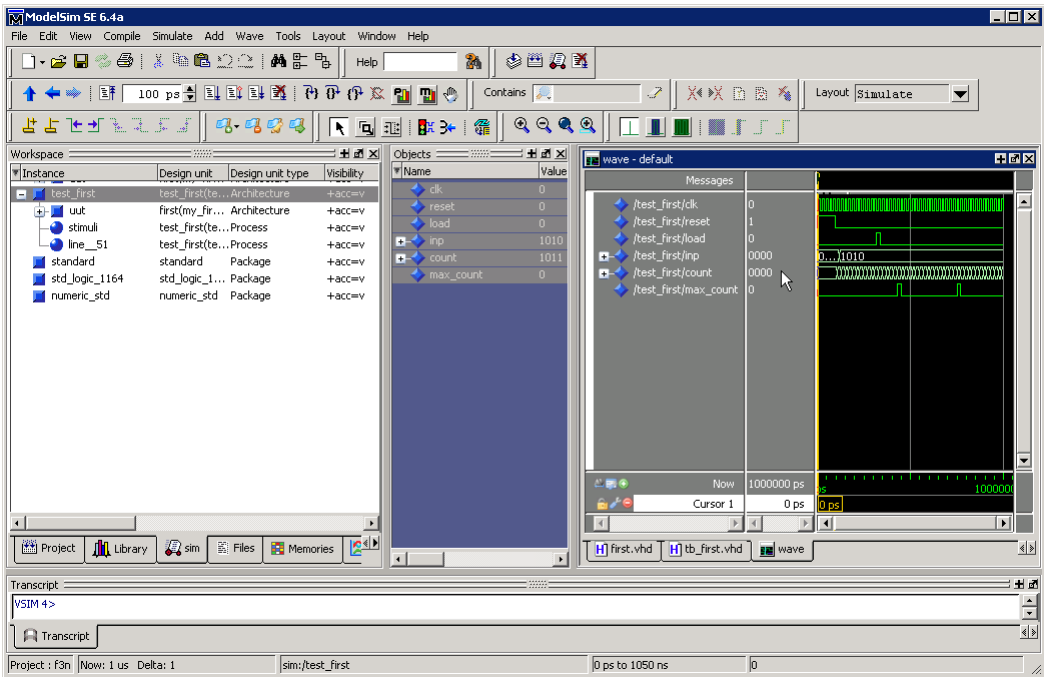
Når designet er lastet inn dukker det opp et "Objects"-vindu. Hvis det ikke dukker opp får man det fram ved å velge **View⇒Debug Windows⇒Objects.**

Fra ”Objects”-vinduet kan man velge hvilke signaler man vil se på under simuleringen. Ved å få fram ”Sim”-vinduet i ”Workspace” og klikke på de forskjellige designenhetene dukker det opp forskjellige ”objects” som man kan velge å se på. Man merker det man vil se på (i dette tilfellet alle objektene) **⇒høyreklikk⇒Add to Wave⇒Selected signals (eventuelt kan man velge Signals in Region)**:



Figur 9. Valg av signaler til Waveform viewer

Etter at man har valgt de signalene man vil observere kan man starte en simulering. Det kan man gjøre ved å taste inn simulatorkommandoen **run (tid)** i konsollvinduet. F.eks. vil **run 1us** simulerere i 1 µs. Taster man **run 1ms** vil man simulere videre 1ms fra tidspunktet man stoppet på ved forrige runkommando. Skriver man **restart** starter man fra tidspunkt 0 igjen, og alle ”waveforms” vil bli nullstillt. Figur 10 under viser signalene fra telleren etter å ha simulert 1 µs.



Figur 10. Waveform-vindu

2.7 Makrofiler (do-filer)

Simulering er en iterativ prosess. Derfor kan det være fornuftig å automatisere store deler av prosessen. Et alternativ til å taste inn eller velge simuleringskommandoer fra menyer er å samle kommandoer i script eller kommandofiler. I Modelsim har man mulighet for å lage script ved hjelp av scriptspråket tcl, eller man kan samle simuleringskommandoer i en såkalt do-fil. En do-fil har gjerne extension `.do`.
F.eks. kan det være fornuftig å spesifisere i do-filen hvilke signaler man ønsker å titte på.
Figur 11 er et eksempel på en do-fil, `sim_first.do`. Man kan lett bygge opp innholdet i en do-fil ved å kopiere fra simulatorens konsollvindu hvor man får ekko av alle kommandoer man har gitt.
Første linjen laster entiteten til testbenken (og underliggende designenheter) inn i simulatoren. De neste linjene legger til signaler i "waveform view"eren, og i siste linjen spesifiserer at man skal simulere 1 μs:

```
vsim work.test_first
add wave sim:/test_first/clk
add wave sim:/test_first/reset
add wave sim:/test_first/load
add wave sim:/test_first/inp
add wave sim:/test_first/count
add wave sim:/test_first/max_count
run 1 us
```

Figur 11. Do-filen sim_first.do

Do-filen i Figur 12 kompilerer de to filene til working library ved navn work. Legg merke til at her er full path inkludert I filnavnet. Hvis man bytter lagringsområde eller PC blir en derfor av og til nødt til å oppdatere do-filen. Det er også mulig å bruke relativ path, men da er det viktig at en står i riktig katalog i Modelsim når man kjører do-filen. En kan finne ut av hvor en står i Modelsim ved å skrive **pwd** i konsollvinduet. Tilsvarende fungerer også kommandoen **cd** for å bytte katalog.

```
vcom -work work -93 -explicit C:/IFI/INF3430/H2005/kokebok/first.vhd
vcom -work work -93 -explicit C:/IFI/INF3430/H2005/kokebok/tb_first.vhd
```

Figur 12. Do-filen comp_first.do

Do-filene kjøres ved å velge **Tools⇒Execute Macro⇒Velg do-fil**. En alternativ måte er å skrive **do filnavn.do** i konsollvinduet. Det er også mulig å samle alle kommandoene i én enkelt do-fil, eventuelt kan en do-fil kalle opp flere andre do-filer.

Det oppfordres til å eksperimentere med simulatoren. Man kan få opp mange flere debuggings vinduer, vi kan sette "breakpoint"s på kodelinjer, og vi kan sette "breakpoint" på signaler når disse får en bestemt verdi og masse annet. Vi kommer innom en del av dette gjennom kurset. Modelsim Xilinx edition er noe begrenset i forhold til Modelsim SE.
Merk at vi har veldig god observerbarhet under RTL-simuleringen. Det er her vi bør utnytte tiden fordi det er her flesteparten av de vanskelige feilene blir oppdaget.

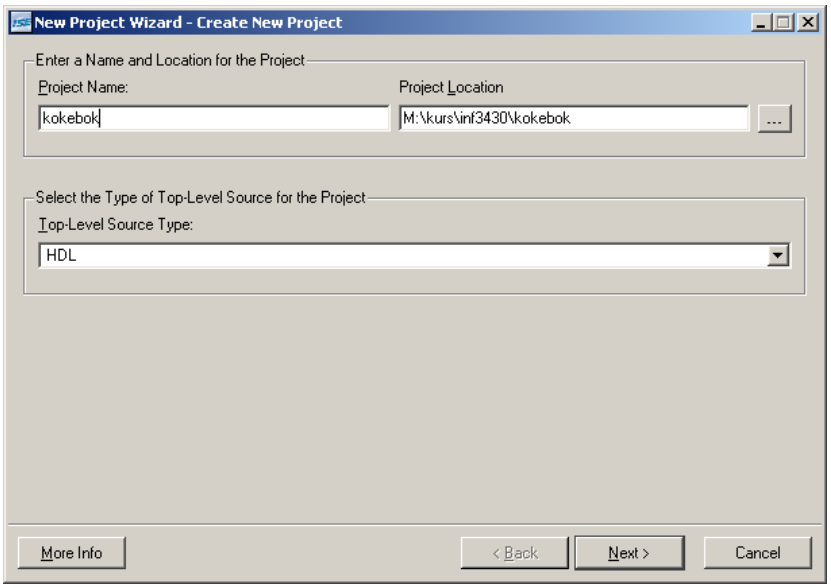
3 Hvordan komme i gang med Xilinx ISE

Xilinx ISE er Xilinx sitt verktøy for Syntese, Place & Route og programmering av disse kretsene. Innenfor hver av disse hovedgruppene av oppgaver finnes det en rekke støtteverktøy. Vi skal i denne oversikten se på de viktigste: Vi skal syntetisere, tilordne pinnenummer, lage timing constraints, kjøre place and route og programmere kretsen.

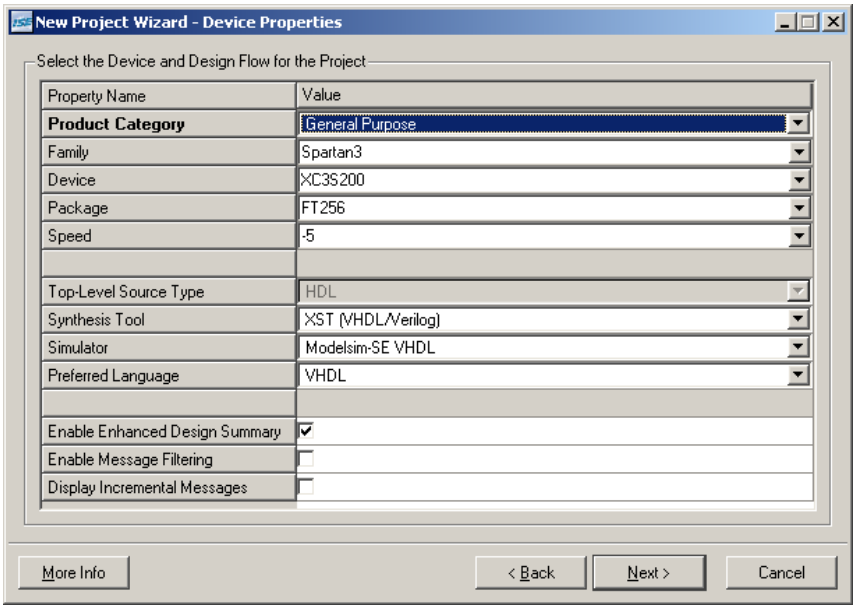
3.1 Oppretting av prosjekt

Xilinx ISE benytter også prosjekter for organisere dataene på en god måte. Vi kan operere med forskjellige prosjekttyper i ISE. F.eks. kan et design være en blanding av skjema og HDL-moduler, eller det kan være et rent HDL-prosjekt. Vi skal se på den sistnevnte typen der alle kildefiler er VHDL-filer.

Vi starter Xilinx ISE ved **Programs⇒Xilinx ISE 9.1i⇒Project Navigator** evt. **Xilinx ISE 9.1i** fra desktopen.

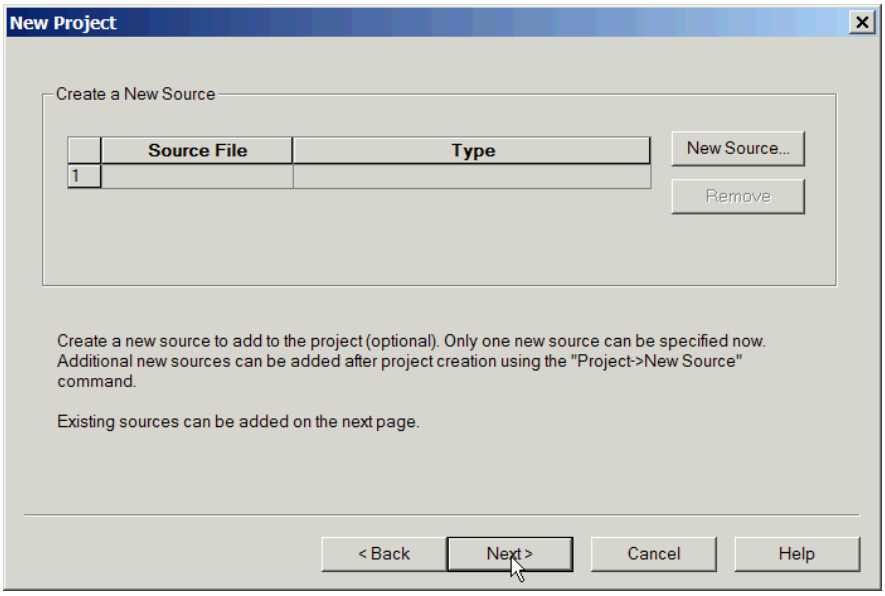


Figur 13. Prosjektoppsett (1). Legg prosjektet til samme katalog som vhdl-filene ligger i, for å unngå problemer senere i kokeboka.

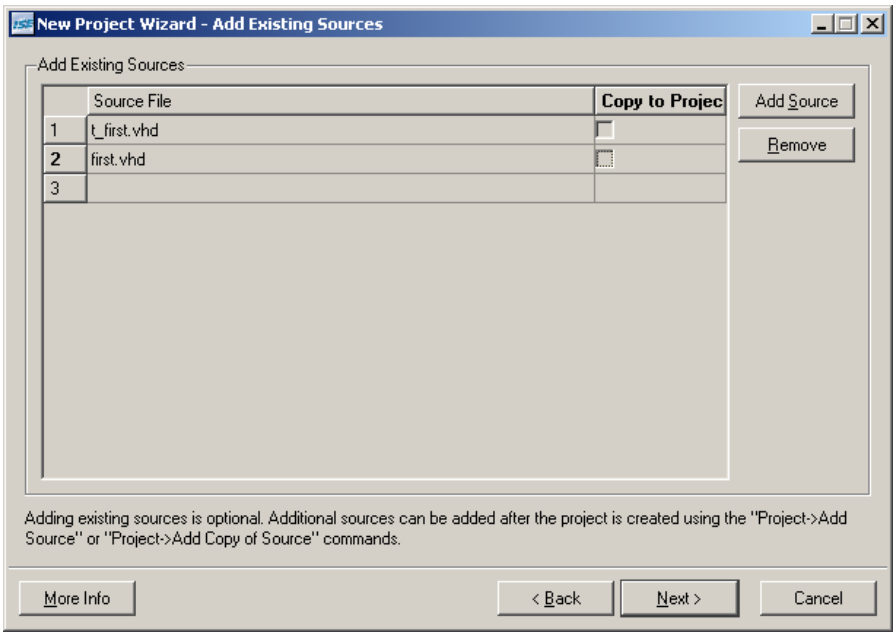


Figur 14. Prosjektoppsett (2)

De valgene man gjør på teknologi og verktøy kan man gjøre om når som helst i designprosessen. I kurset så er imidlertid teknologivalget gjort. På labkortet sitter det en Spartan-3 XC3S200FT256 FPGA. FPGA-ene er sortert i henhold til hvor raske de er, høyere verdi på speedgrade betyr raskere krets.

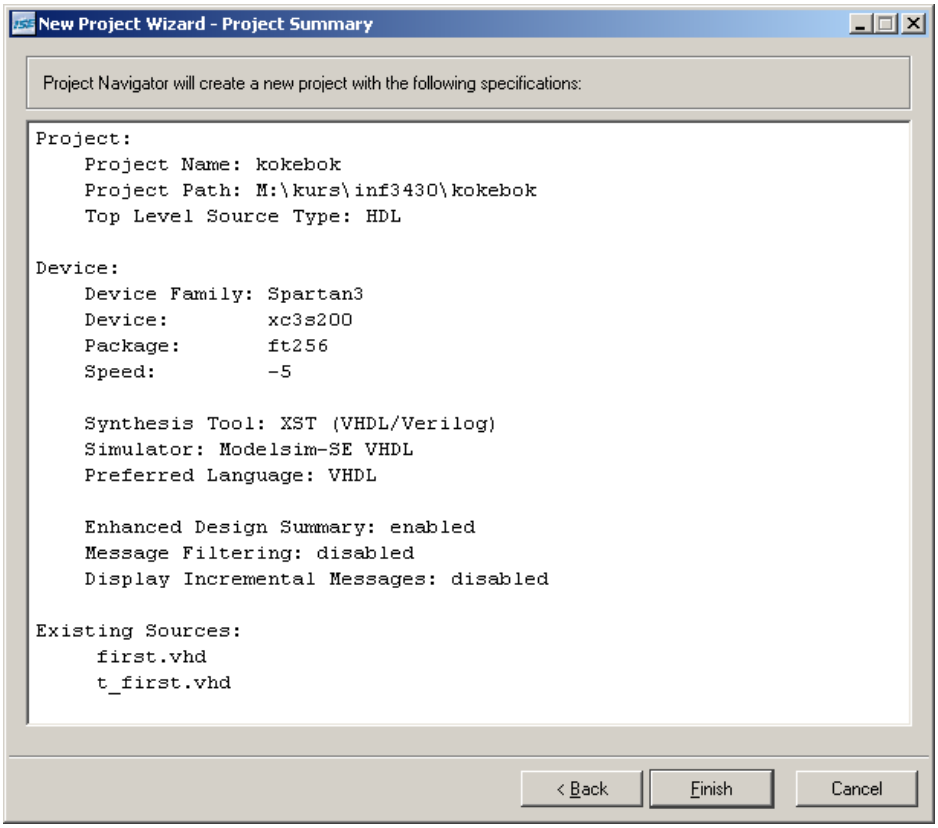


Figur 15. Lage nye kildefiler. Hopp over denne.



Figur 16. Legg til eksisterende kildefiler

Som for Modelsim kan vi velge om vi vil lage en kopi av kildefil eller referere til den/de fra opprinnelig lokasjon. Det er opp til den enkelte hva man vil gjøre her med dette, det viktigste er man har et aktivt forhold til hvordan man vil organisere sine filer.



Figur 17. Prosjektinformasjon

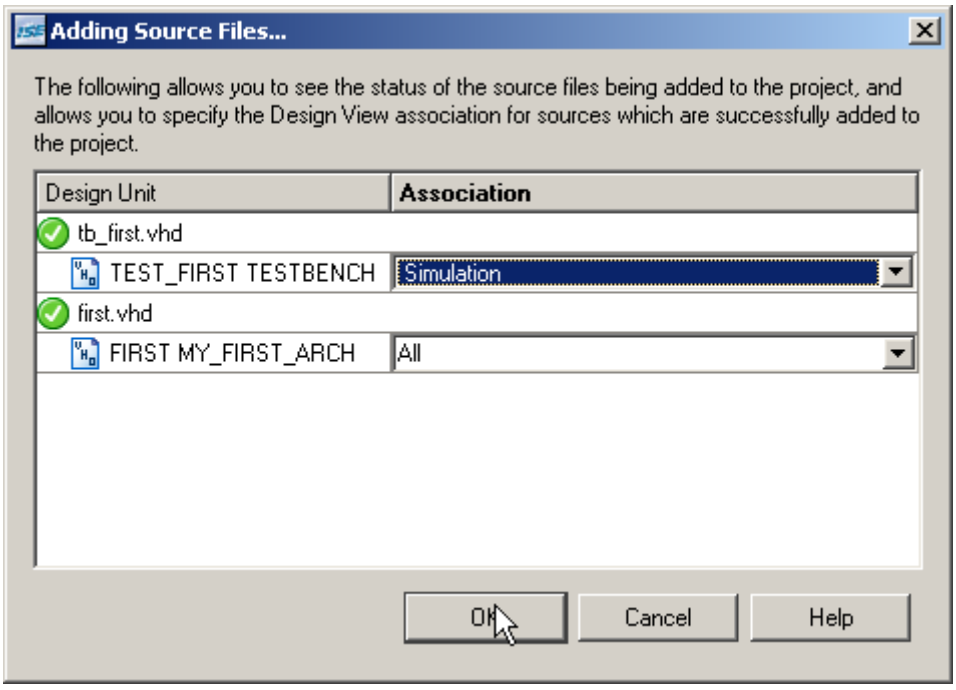


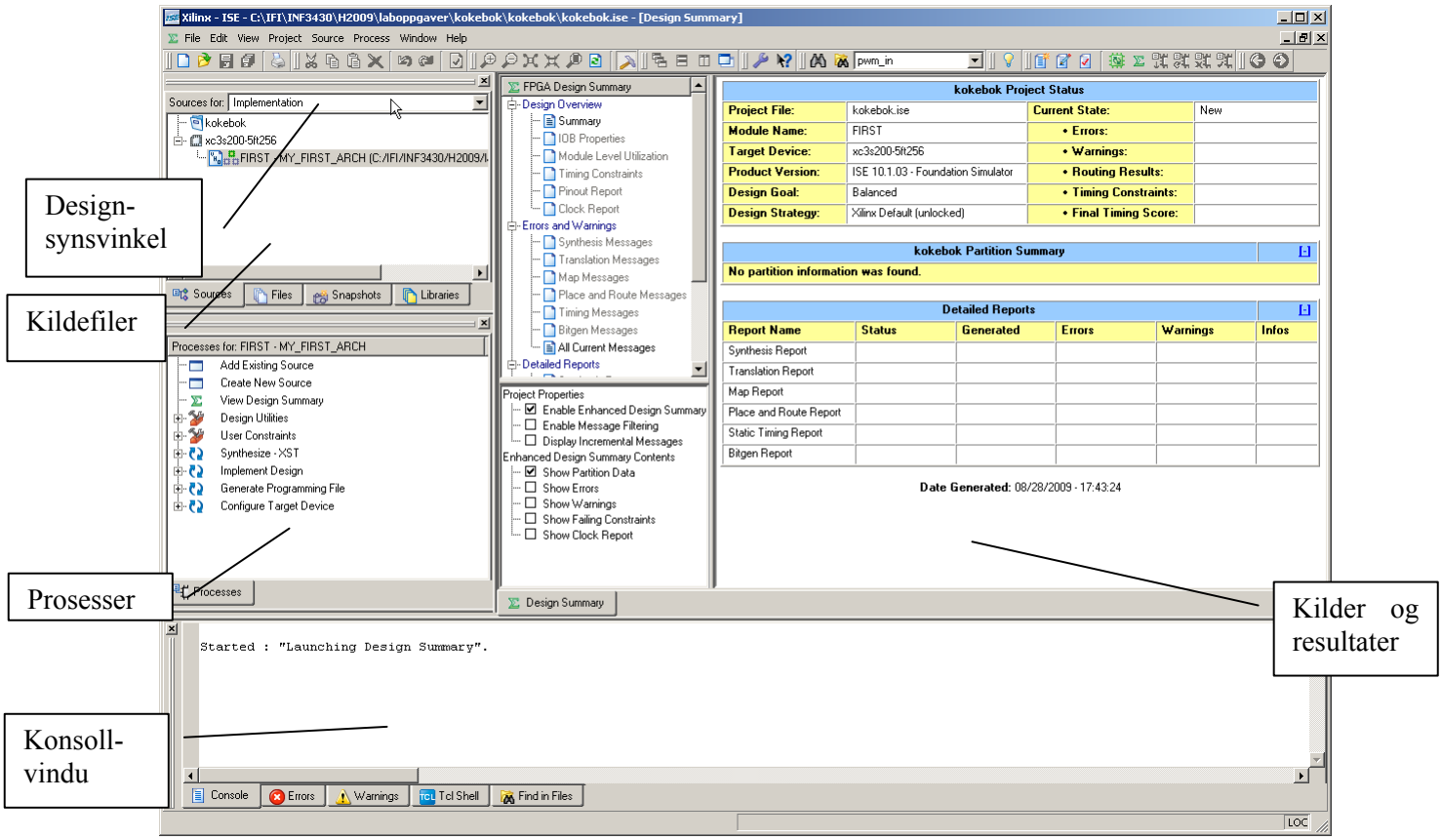
Figure 18. Kildefiltyper

ISE spør om hva slags rolle de forskjellige filene skal ha. Denne informasjonen blir brukt av ISE til å sette opp de forskjellige designsynsvinklene (”Design View”) i brukergrensesnittet.

3.2 Bruk av ISE

Når man har opprettet prosjektet eller man starter "Project Navigator" neste gang får man opp et skjermbilde som i figuren under. Bildet er organisert i fire deler:

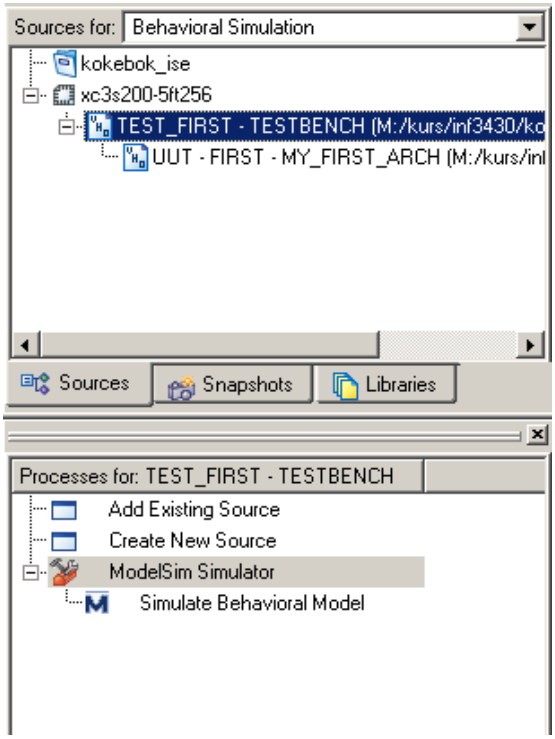
- Kilder (Sources)
- Prosesser
- Consol (program output (stdout))
- Kilder/resultater



Figur 19. Project navigator

Legg merke til at man får se forskjellige sett av filer ved å velge synsvinkel i "Sources for:"-menyen. Avhengig av hvilken synsvinkel man er i får man opp forskjellige prosesser for filene.

Ved å velge "Behavioral Simulation" som synsvinkel får man opp følgende prosesser for test_first:



Figur 20. Prosess for testbenk

Vi skal nærmere på simuleringer etter vi har implementert designet (se avsnitt 3.7).

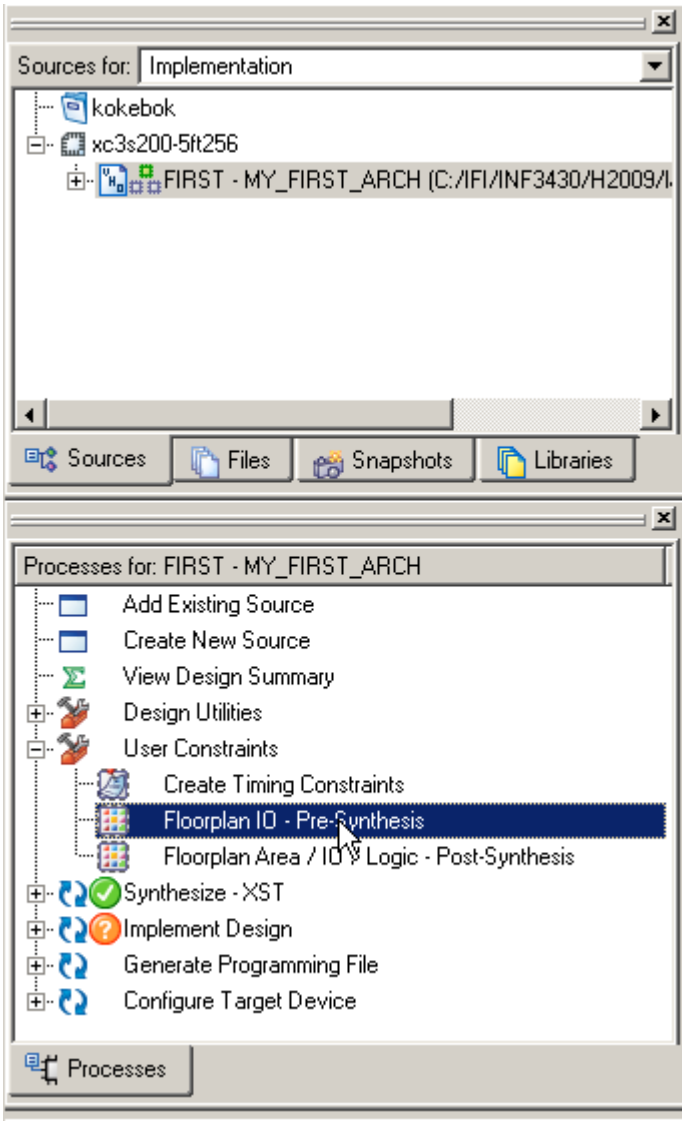
3.3 Tilordning av pinner (Pin number constraints)

Ideelt sett skulle det bare være å sette i gang en syntese og deretter place & route.
Men vi må i høyeste grad forholde oss til ***at designet skal implementeres på en krets som står montert på et ferdig kort. Det betyr at all pinneplassering er låst fast***.
At pinneplassering er gjort på forhånd er ofte situasjonen i virkeligheten, fordi man gjerne ønsker at utlegg av kort og utvikling av innmaten i FPGA-ene skal være parallelle aktiviteter for å spare tid i prosjektet. En ugunstig pinneplassering kan gå ut over ytelse.
Siden pinneplasseringen er gitt for oss må vi kontrollere dette. Dette gjøres ved å bruke en "user constraint file". Den skal ha extension `.ucf`. I UCF-filen kan man bl.a. tilordne toppnivåentiteten riktige pinnenummer. UCF-filen er en tekstfil som kan redigeres direkte i teksteditoren i ISE. Skal vi lage "pin number constraints" eller areal-"constraints" kan vi redigere disse gjennom bruk av et verktøy som heter PACE. I vårt tilfelle skal vi tilordne signalene til entiteten FIRST til pinnenummere:

Tabell 1. Pinnetilordning

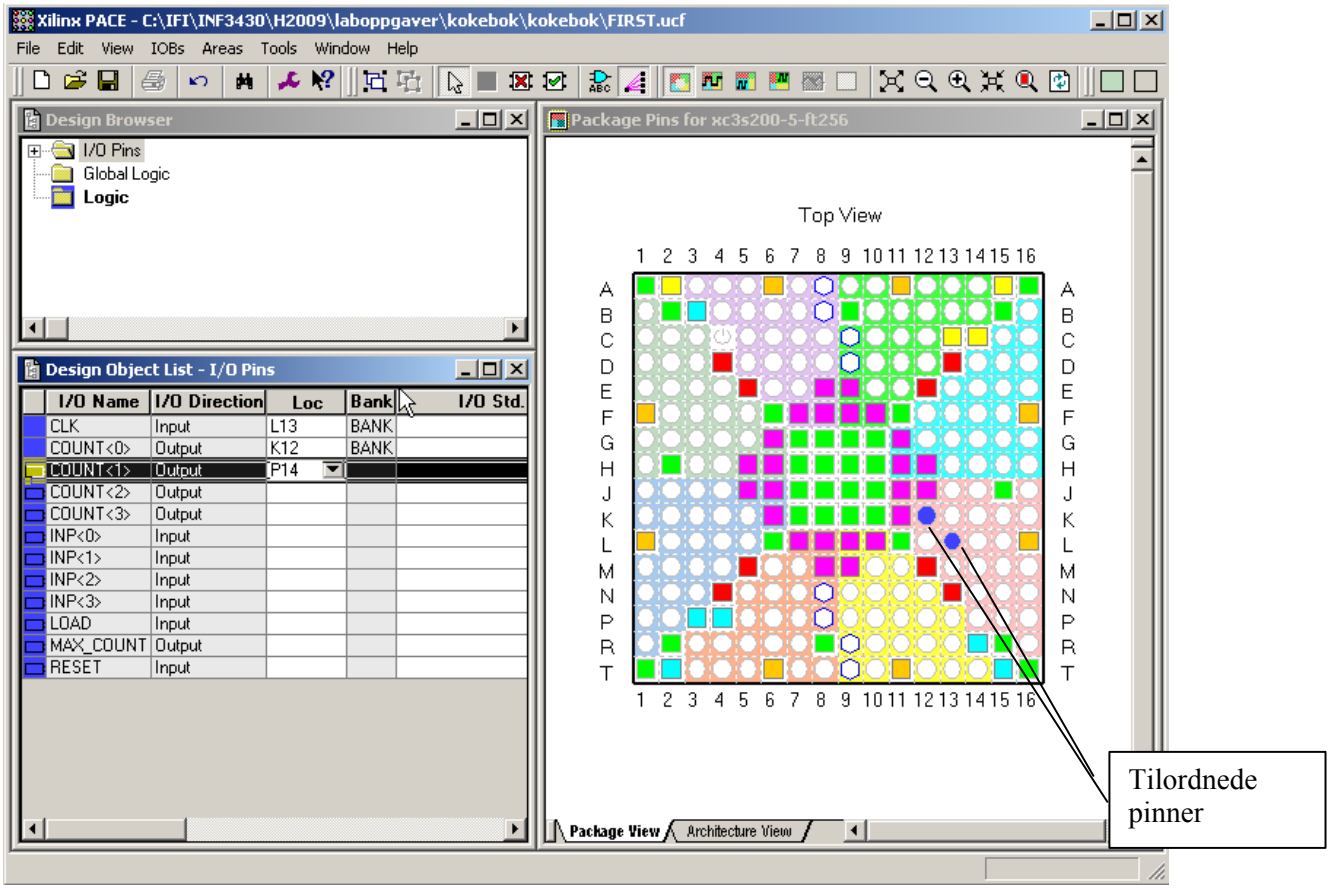
Signal	**Pinnenummer**	**Signalnavn på kortet**
CLK	L13	BTN2
RESET	L14	BTN3
LOAD	K13	SW7
INP (0)	F12	SW0
INP (1)	G12	SW1
INP (2)	H14	SW2
INP(3)	H13	SW3
COUNT(0)	K12	LD0
COUNT(1)	P14	LD1
COUNT(2)	L12	LD2
COUNT(3)	N14	LD3
MAX_COUNT	P11	LD7

Merk first.vhd og ekspander "User Constraints" i "Process-vinduet". Velg **"Floorplab I/O – Pre-Synthesis"⇒høyreklikk⇒Run**.



Figur 21. User constraints

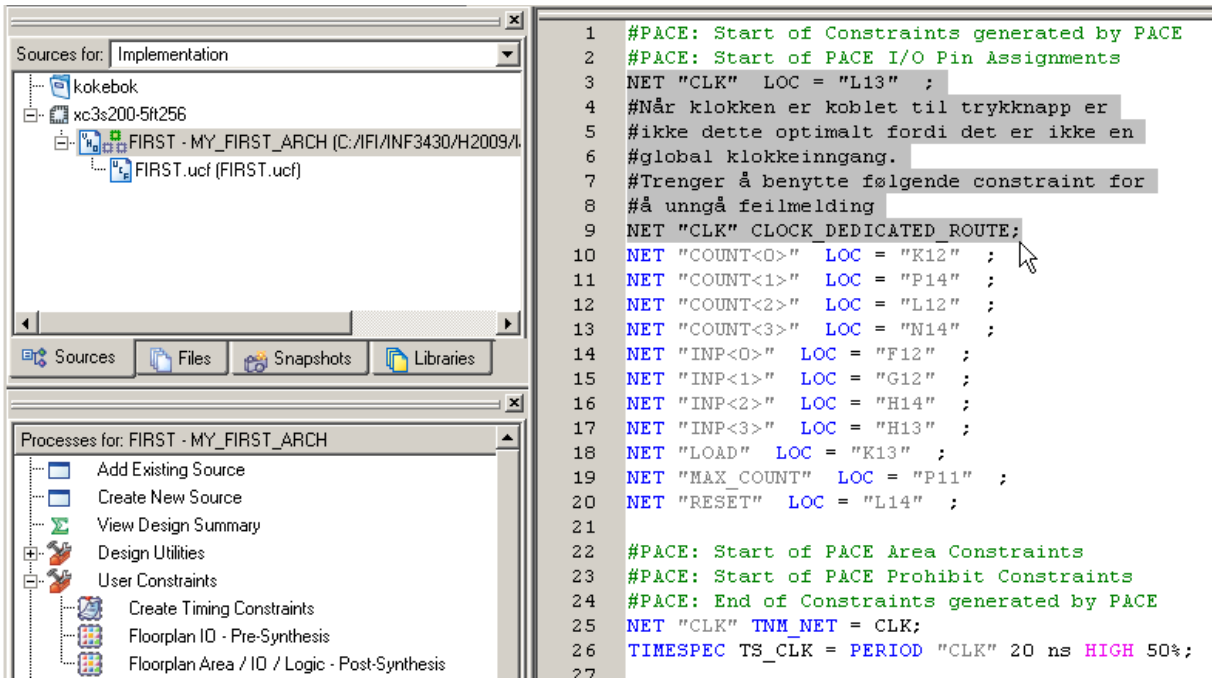
Svar ja dersom du får valget om å opprette en ucf-fil (Her er det noen som har hatt problemer hvis ikke toppnivå-VHDL-filen ligger i samme katalog som prosjektfilene). Dersom det kommer opp en melding om at man kan benytte programmet PlanAhead til pinneplassering trykker man OK, og krysser eventuelt av for man ikke vil se denne meldingen igjen. Man blir styrt inn i programmet PACE som man benytter til å lage pinnenummer-"constraints" og areal-constraints. Vi skal se på pinnenummer-"constraints". Man kan nå tilordne pinner i henhold til Tabell 1:



Figur 22. PACE-Tilordning av pinner (Package view)

Når man er ferdig med å legge inn pinnenummer "constraints" velg **File⇒Save**. Du får kanskje spørsmål om "bus-delimiter". Velg XST Default <>.

Velger man **Edit Constraints (Text)⇒høyreklikk⇒Run** åpnes ucf-filen for redigering med ISEs interne teksteditor (dette gjelder dersom ucf-filen eksisterer og er lagt til prosjektet):



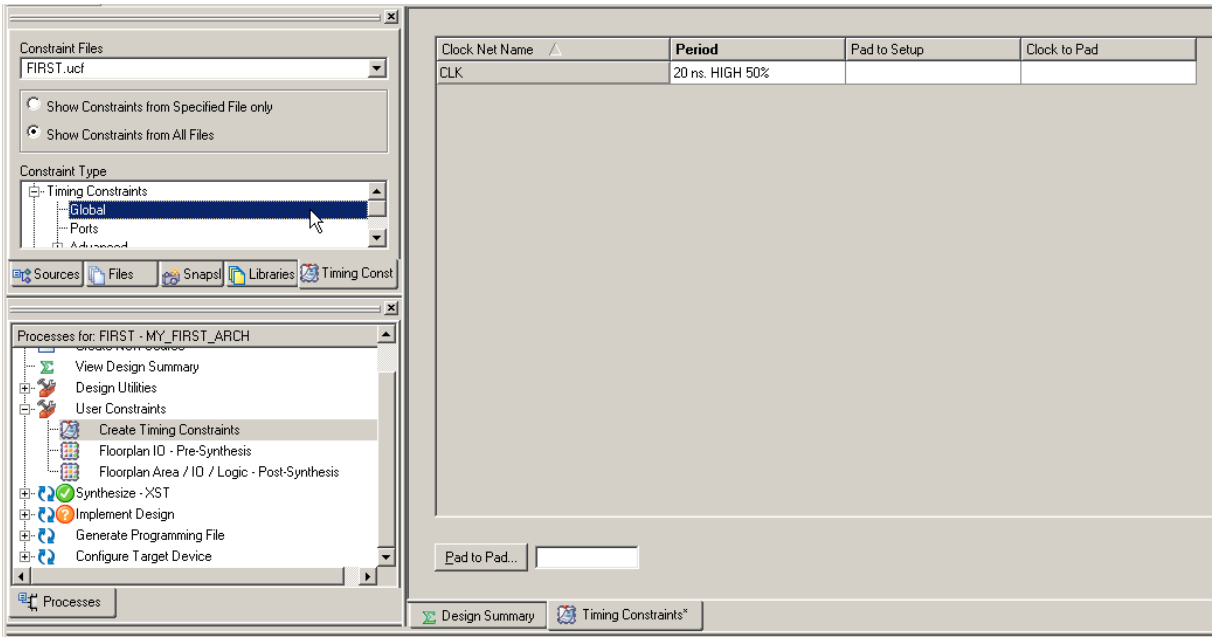
Figur 23. UCF-fil (User Constraint File)

Legg merke til at dette tilfellet tilordner vi klokken til en trykknapp. Denne pinnen er ikke et global klokkeinngang og følgelig ikke optimal å bruke som klokkeinngang. Nåværende versjon av ISE (10.1i SP3) gir feilmelding på dette. For å komme rundt dette må man legge til følgende constraint i ucf-filen: NET "CLK" CLOCK_DEDICATED_ROUTE; I senere oppgaver skal vi benytte 50MHz crystal oscillator som klokke. Denne er koblet til en global klokkeinngang.

3.4 Timing constraints

Etter at man er ferdig med pinnetilordning kan man legge til andre "constraints", f.eks. "timing constraints". Vi skal nå nøye oss med å legge til et "timing constraint", nemlig perioden på CLK. På kortet er det en klokkeoscillator på 50MHz, så det er naturlig å velge en periodetid på 20 (ns). Vi velger **"Create Timing Constraints"⇒høyreklikk⇒Run**. Da åpnes et program som heter "Xilinx Constraints Editor". Tast in 20 for perioden og velg **File⇒Save**. Dette medfører at UCF-filen blir oppdatert med nye "constraints". Syntesen og place and route verktøyene vil nå forsøke å tilfredsstille disse "constraints" vi har lagt inn. Legg merke til at UCF-filen er "constraints" til Place and Route-verktøyet og ikke til syntheseverktøyet.

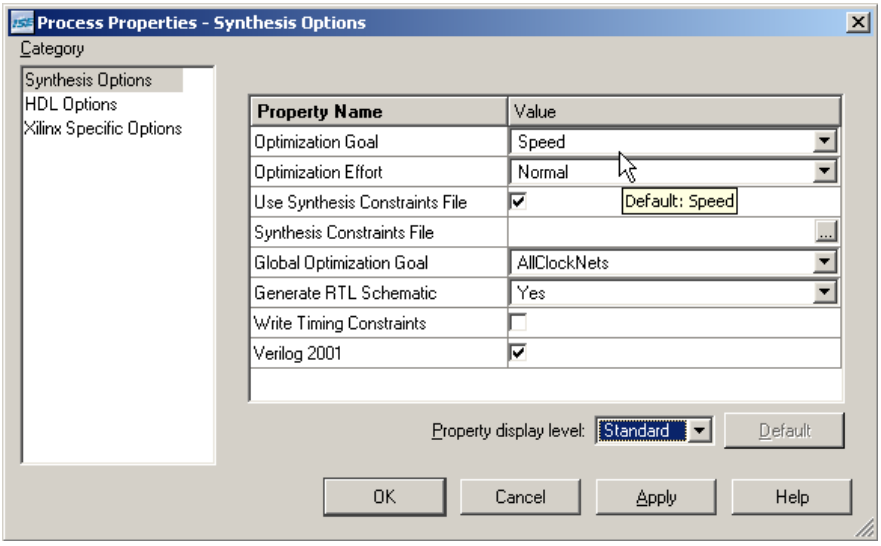
Vi er nå klare til å se nærmere på synteseprosessen.



Figur 24. Constraints editor

3.5 Syntesen

Vi kan også påvirke syntesen ved å benytte "constraints". I prosessvinduet merker vi **Synthesize-XST⇒Høyreklikk⇒Properties** og får fram følgende vindu:



Figur 25. Syntese constraints

Vi lar syntese-constraints være som de er foreløpig og kommer tilbake til de senere i kurset.

Merker vi **Synthesize-XST⇒Høyreklikk⇒Run** starter synteseprosessen. Resultatet av syntesen er en gatelevel-nettliste, dvs. sammenkobling av basiselementer. Syntesen består av flere deltrinn: Først en RTL-syntese der boolske ligninger og flip-flop'er blir dannet på en mer generelt format. Under Low-level synthesis blir designet optimalisert i forhold til den aktuelle målteknologien.
Vi får oppsummert resultatet av syntesen i en egen rapportfil. Denne innholder bl.a. statistikk over hva slags basiselementer som er benyttet og estimat over ytelse. Man kan se på denne i "Design Summary" vinduet og velge "Synthesis Report". Man vil også få fram kortfattet og viktig statistikk i "Design Summary" vinduet.
Vi kan også få generert koblingsskjema ut fra RTL-syntesen og et skjema generert på basis av aktuell målteknologi.

3.6 Design implementation

Etter syntesen kan vi gå over til å implementere designet i den valgte kretsen.
Implementasjonen består av tre deltrinn

- Translate
- Map
- Place and Route

3.6.1 Translate

I translate fasen blir nettlisten som er laget av synteseverktøyet oversatt til Xilinx interne nettlisteformat.

3.6.2 Map

I mappingfasen blir basiselementene tilordnet til basis byggeklosser i den aktuelle kretsen. Pinner blir også tilordnet på dette stadiet.

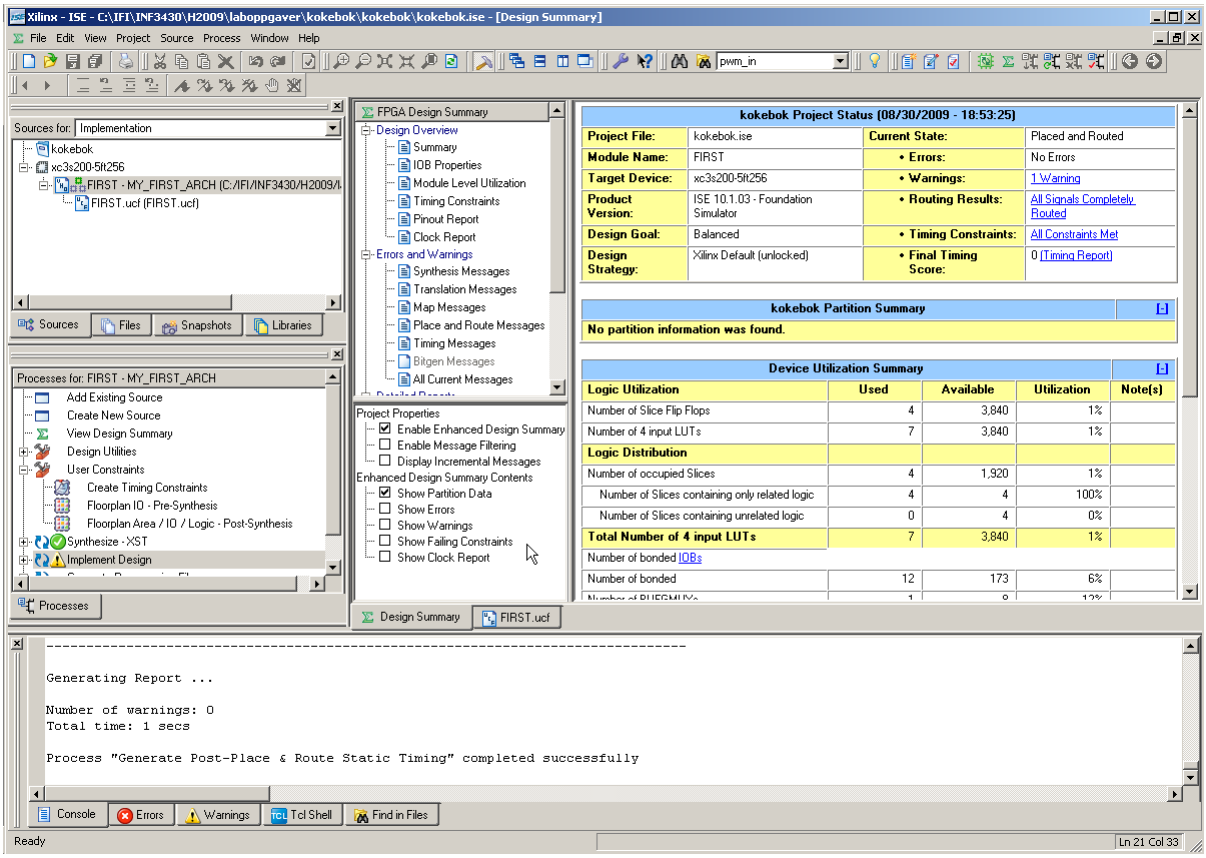
3.6.3 Place & Route

Først blir basis byggeklossene som ble mappet plassert rundt i kretsen. "Timing" og "pin constraints" (og ikke minst "area constraints") er viktige for at denne prosessen gir resultat som bidrar til at "constraint"ene blir oppfyllt.

Etter ”place” prosessen går man over til ”route” fasen der de forskjellige basiselementene blir koblet sammen. ”Timing constraint”ene blir her flittig benyttet slik at tidsforsinkelser i sammenkoblinger ikke overstiger ”constraint”ene.

Vi starter implementasjonen ved å gå inn i prosessvinduet og merke ”Implement design”⇒Run. Da vil alle tre delprosessene bli gjennomført (dersom det ikke framkommer feil på veien).

Etter at implementasjonsprosessen er ferdig får vi dannet en rekke rapporter. Vi kan velge å se på disse ut fra ”Design Summary” vinduet:



Figur 26. Design summary etter implementert design

3.7 Videre simuleringer

Man kan simulere på flere nivåer etter RTL-nivået:

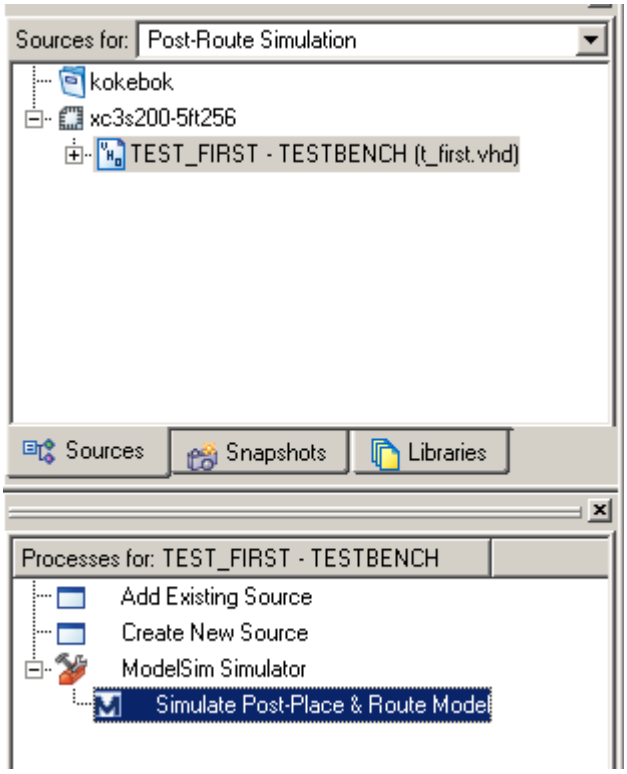
1. Post Syntese
2. Post Translation
3. Post Map
4. Post Route

1. og 2 er i prinsippet likeverdige simuleringer som verifiserer at syntesen har gitt korrekt resultat. Forskjellen ligger i hvordan kretsen er modellert. 1. er modellert ut fra Xilinx UNISIM biblioteket, mens 2, 3 og 4 er modellert ved bruk av VITAL biblioteket. Vi skal se på nærmere på UNISIM og VITAL bibliotekene senere i kurset. Når designet blir stort kan simuleringer ved bruk av VITAL være ganske tid- og ressurskrevende(RAM).

Man kan starte de forskjellige typer simulering automatisk innenfra ISE eller manuelt i Modelsim
Vi skal se på både den automatiske og den manuelle oppstarten.

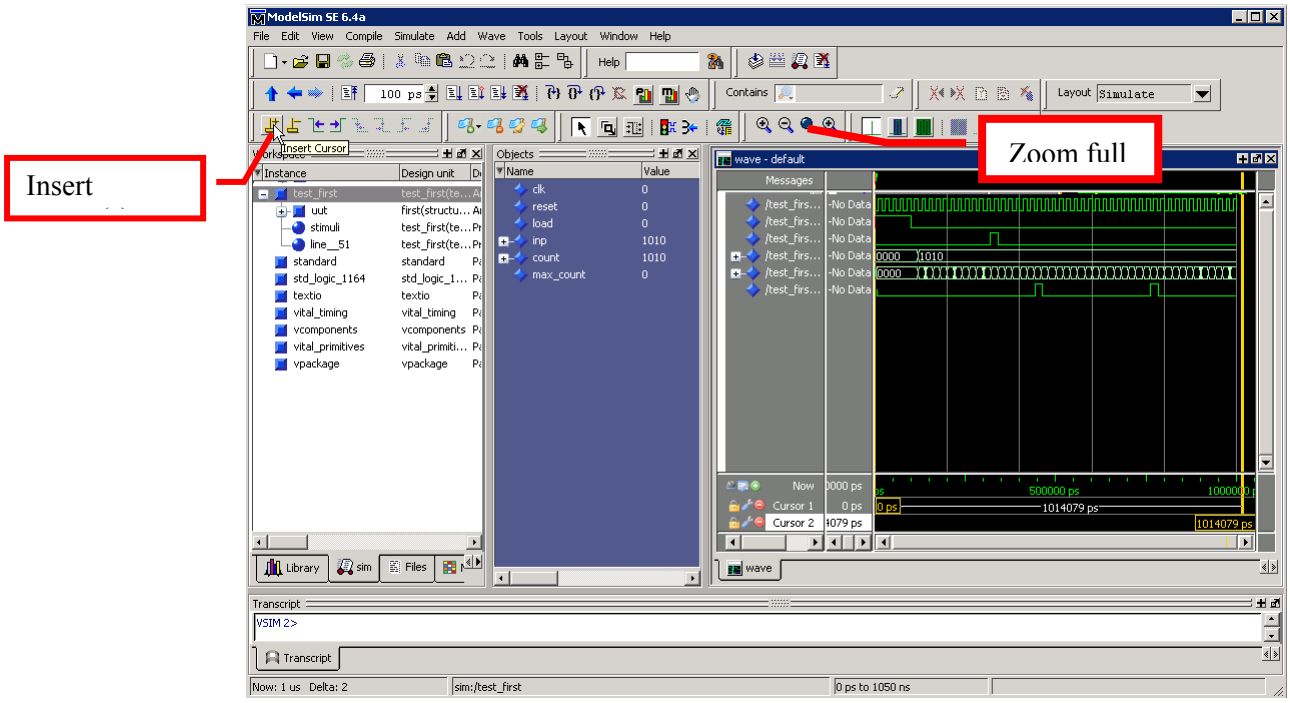
3.7.1 Automatisert oppstart av Modelsim

Man kan starte de forskjellige simuleringene ved å velge riktig synsvinkel, og deretter merke testbenkfilen:



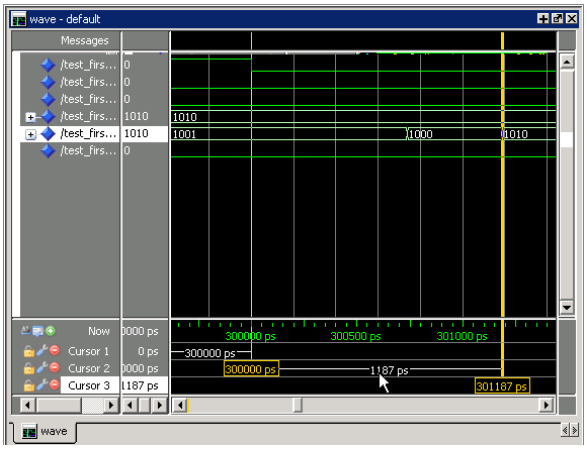
Figur 27. Oppstart av Post Place & Route-simulering

Det genereres automatisk do-filer som kompilerer, laster designet inn i simulatoren, velger ut signaler til entiteten vi skal simulere, og kjører simulering en viss tid. Figur 28 viser et bilde fra en Post Place & Route simulering:



Figur 28. Post Place & Route-simulering

Figur 29 viser et forstørret utsnitt og der vi har lagt inn to kursorer (markører) for å måle tidsforsinkelser:



Figur 29. Bruk av kursorer for å måle tidsforsinkelser

3.7.2 Manuell oppstart av Modelsim

ISE kan generere simuleringsmodeller for de forskjellige fasene på en manuell på måte.
F.eks. kan man generere en postsyntesefil ved å gå inn i synteseprosessen og merke **"Generate Post-Synthesis Simulation Model"⇒Run.**
Tilsvarende kan man gjøre i Translate, Map og Place & Route prosessene. Man får da generert følgende filer:

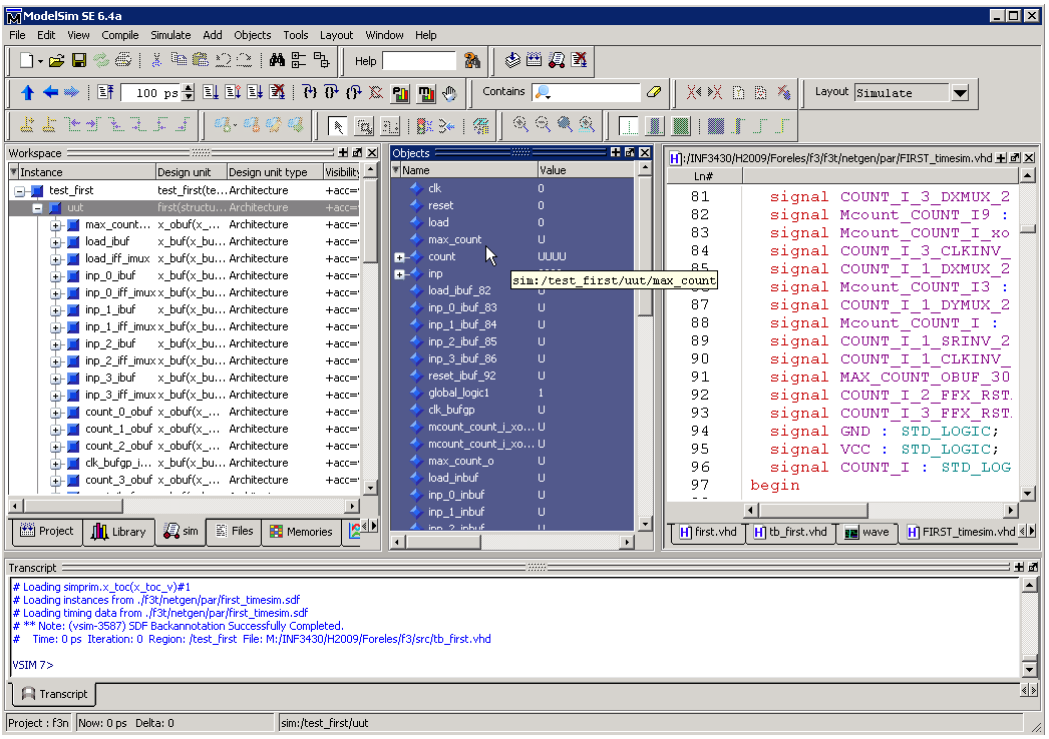
Tabell 2. Genererte VHDL-modeller

Prosess	**VHDL-fil generert**
Syntese	<top-nivå entitet>_synthesis.vhd Eksempel: first_synthesis.vhd
Translate	<top-nivå entitet>_translate.vhd Eksempel: first_translate.vhd
Map	<top-nivå entitet>_map.vhd Eksempel: first_map.vhd
Place & Route	<top-nivå entitet>_timesim.vhd <top-nivå entitet>_timesim.sdf Eksempel: first_timesim.vhd first_timesim.sdf

Etter å ha generert disse simuleringsmodellene kan man gå inn i Modelsim prosjektet som ble dannet i avsnitt 2.1.

Når vi skal kompilere så kan vi velge å kompilere modellene til biblioteket work. Vi har da potensielt 4 vhdl-filer med samme "entitet", men med forskjellige "architecture". Spørsmålet blir: hvem benytter testbenken seg av? Svar: Den sist kompilerte. Vi sier at testbenken binder seg til den sist kompilerte "architecture". Dette kalles "default binding". Vi kompilerer modellene til work ut fra denne forutsetningen og passer på å kompilere den modellen vi ønsker å simulere sist. Vi skal senere lære om "configurations" i VHDL som gir oss veldefinerte mekanismer for å håndtere "entitet"er som

har flere ”architcture”. Vi skal også lære å håndtere flere biblioteker enn work og benytte disse som working library.
Prøv å simulere ved bruk av de forskjellige modellene. Gå inn i ”Sim”-vinduet i workspace og ”browse” nedover i UUT. Figur 30. Postsyntese simulering nedenfor viser et utsnitt fra postsyntese simuleringen:



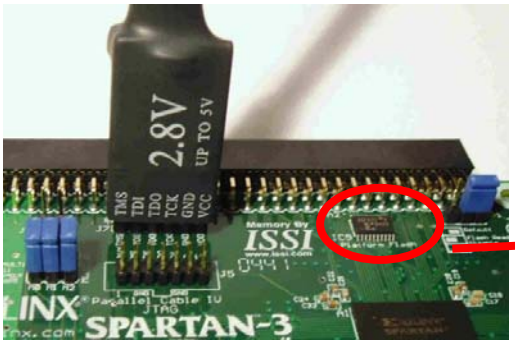
Figur 30. Postsyntese simulering

For å foreta en timingsimulering må man utføre noe vi kaller ”SDF backannotation”. Dette får vi til ved gi følgende kommando i konsollvinduet:

```
vsim -lib work -sdfmax /UUT=<path eller relative path>first_timesim.sdf -t 1ps test_first
```

Vi skal komme tilbake til mer detaljer om VITAL standarden og Standard Delay Format filer (SDF-filer) senere i kurset.
Nå kan dere simulere med de forskjellige modellene og studere ”waveforms” for å se om det er samsvar.

3.8 Device Programming



Figur 31. Programmeringskabelen

3.8.1 Tilkoblinger

Man benytter et eksternt minne for å lagre konfigurasjonen til en FPGA. For test og debug kan det være hensiktsmessig å laste designet rett inn i kretsen uten å måtte programmere et eksternt minne.
Alle Xilinx FPGA-er og CPLD-er kan programmeres via JTAG. JTAG, eller IEEE1149.1 Boundary Scan, er egentlig en standard som er utviklet for å teste forbindelser på et kretskort ved å skifte bits gjennom et langt skiftregister, som går gjennom flere kretser. JTAG er etter hvert blitt et vanlig programmeringsgrensesnitt for FPGA-er, CPLD-er og mikrokontrollere. Vi skal se nærmere på JTAG senere i kurset. Se forøvrig kapittel 5 i Maxfield og http://www.jtag.com.

På testkortet er to kretser forbundet i en såkalt JTAG-kjede: Spartan-3 FPGA og en Flash EPROM som normalt skal lagre FPGA-konfigurasjonen. Vi kan programmere FPGA eller konfigurasjonsminnet ved å benytte en spesielt tilpasset programmeringskabel. Programmeringskabelen (fra Digilent) vi skal benytte kobles til parallellporten på PC-en. (Det finnes nyere kabler som benytter USB).

3.8.2 Nedlasting av konfigurasjon direkte til FPGA via JTAG

Før man lager en programmeringsfil må man sette ned konfigurasjonshastigheten. Dette gjøres på følgende måte: **Generate Programming file⇒høyreklikk⇒ Properties**. Velg **Configuration Options⇒Configuration Rate⇒3.**

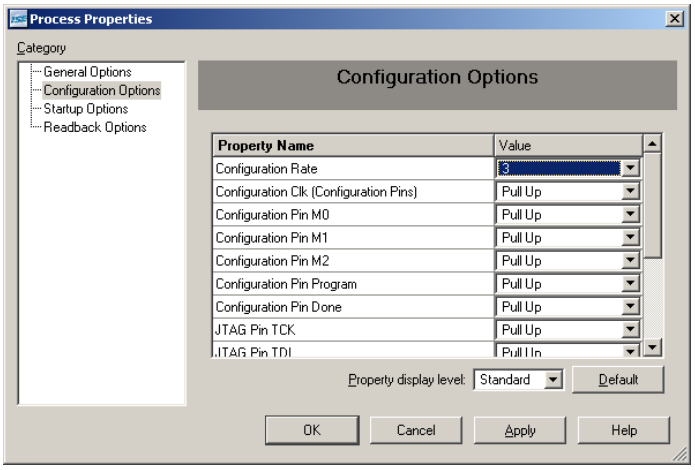
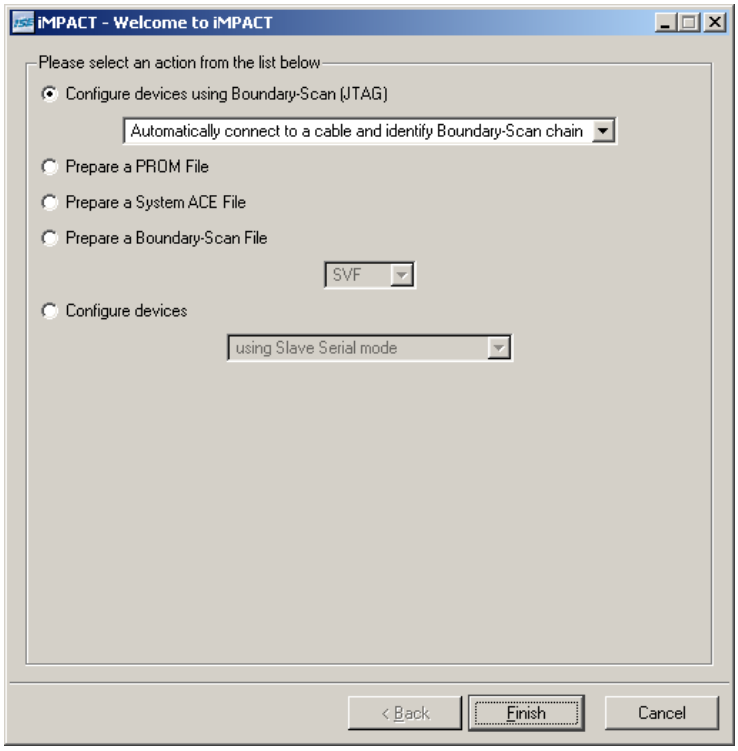


Figure 32. Configuration rate

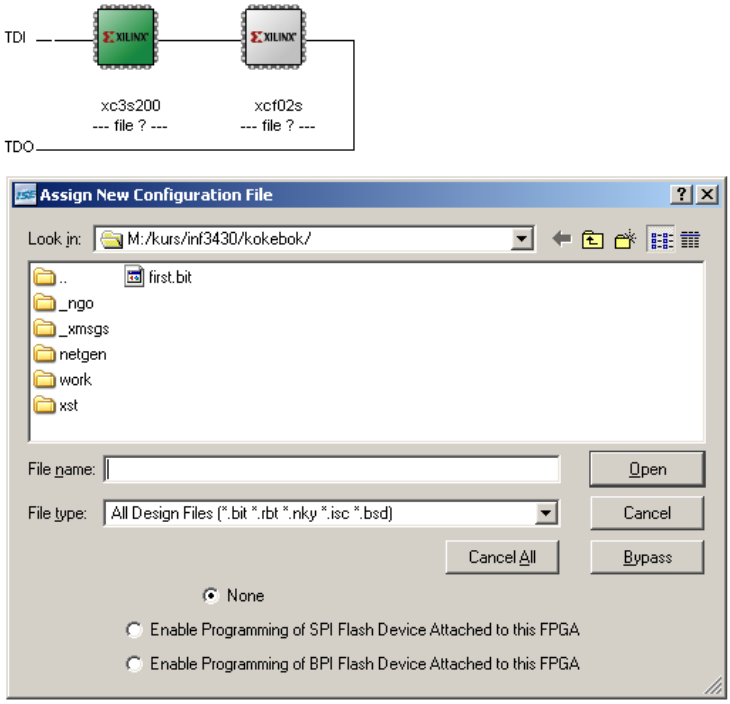
Koble så til programmeringskabelen, sett strøm på testkortet og start prosessen **"Configure Device (iMPACT)":**

Figur 33: Configure Device

Første gangen man programmerer får man opp valgene i Figur 34. Etter man har programmert en gang kommer man rett inn i et skjermbilde som vist i Figur 35.
Velg "**Configure devices using Boundary-Scan(JTAG)**" i Figur 34.

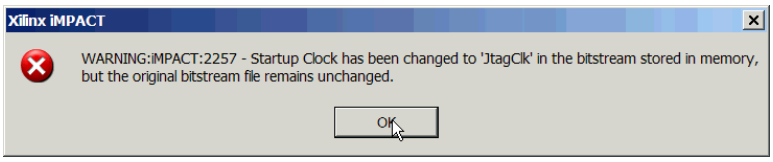


Figur 34



Figur 35

Programmeringsfilen (bitstream) for FPGA-en ble generert da vi startet iMPACT-prosessen, og heter first.bit. Så i neste trinn assosierer vi denne filen med FPGA-en.
Vi får advarselen som Figur 36 men det er helt normalt. Under normal drift konfigureres FPGA-en ved at den genererer klokken CCLK som benyttes til å lese fra et eksternt minne. Kilden til oppstartsklokken (startupclk) velges ut fra en av de første bitene av bitstream-filen. Siden vi skal laste via JTAG benyttes JTAG klokken til innlasting av bitstream'en fra PC-en.



Figur 36

For å bli kvitt denne advarselen kan vi merke **Generate Programming file⇒høyreklikk⇒ Properties**. Velg **Startup Options⇒FPGA Startup Clock⇒JTAG Clock**:

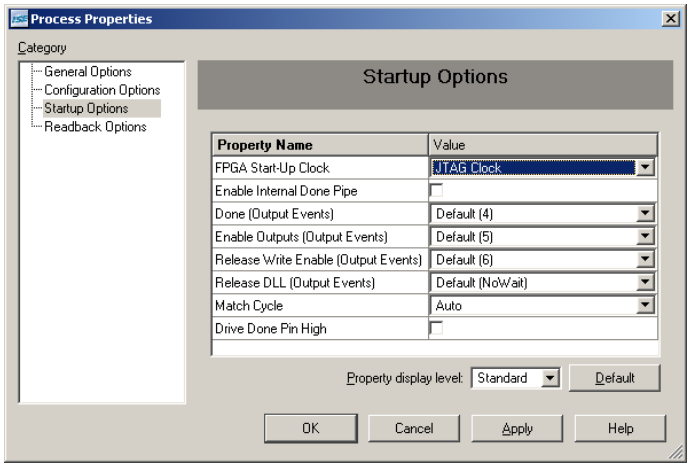
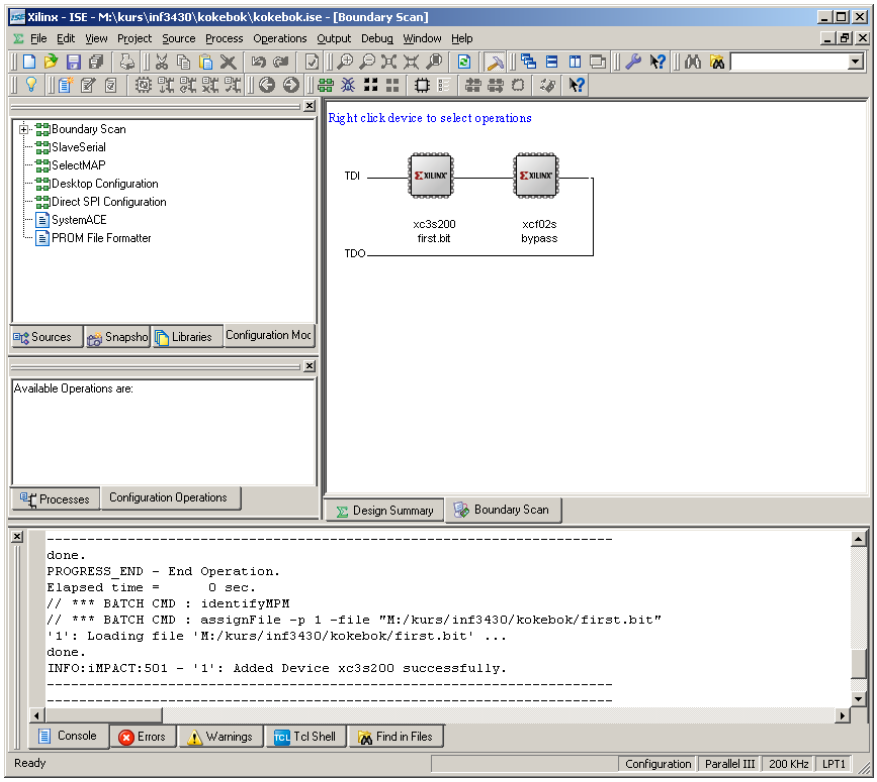


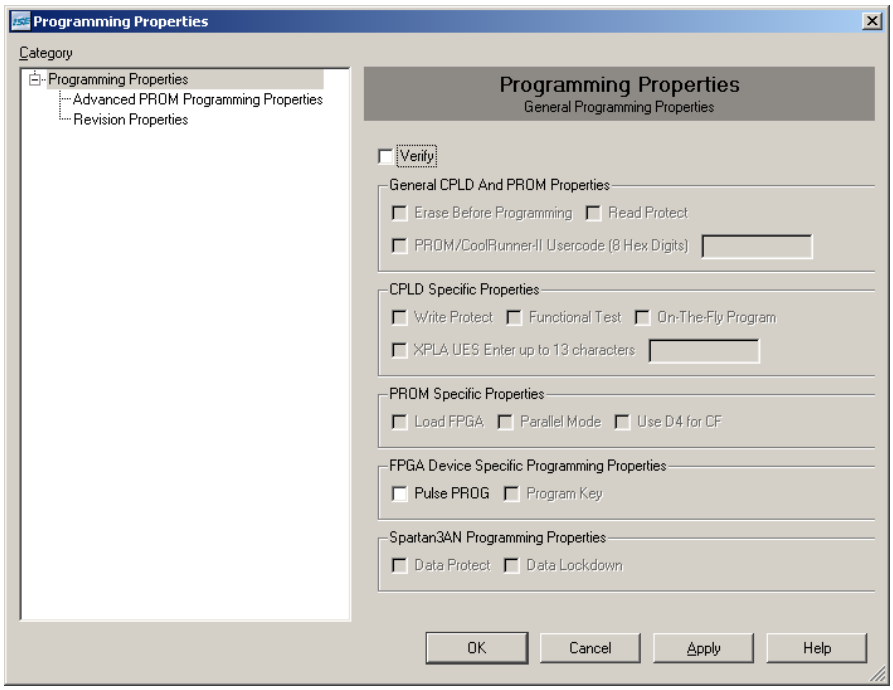
Figure 37. Valg av Start-up clock

Velg BYPASS når det kommer spørsmål om programmeringsfil for konfigurasjonsminnet.



Figur 38. iMPACT programmeringsvindu

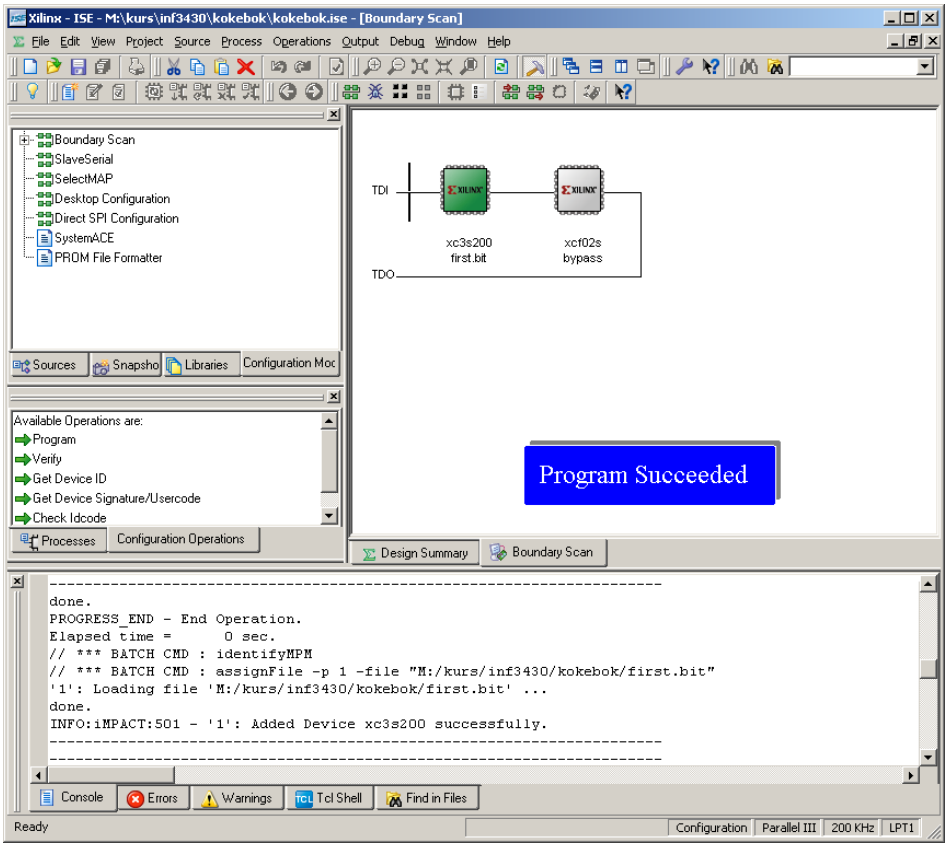
Merk **xc3s200⇒høyreklikk⇒Program**. Ikke kryss av Verify



Figur 39

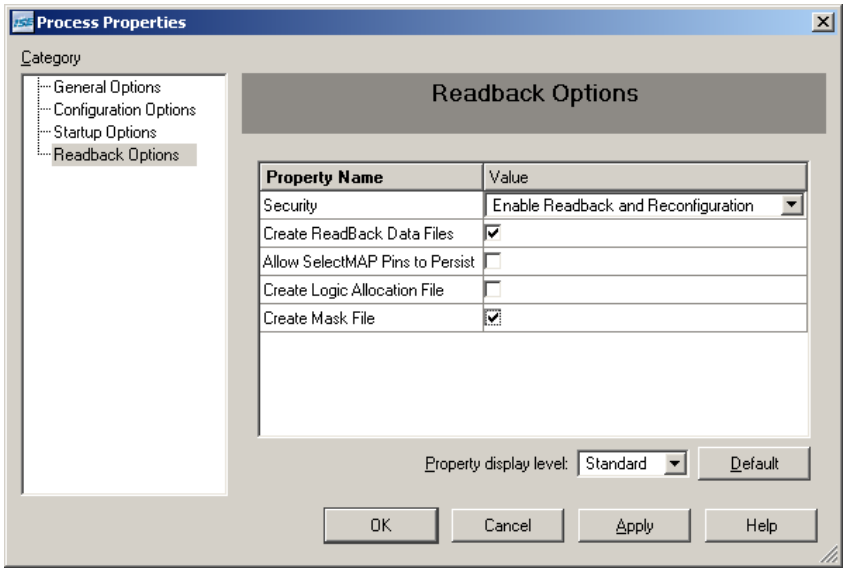
Det kan av og til oppstå problemer med å få til en vellykket programmering av kortet. Det er ikke sikkert hva dette skyldes, men i noen tilfeller hjelper det å bare prøve på nytt noen ganger til det fungerer. Det kan også være lurt å forsikre seg om at kabeltype er satt til "**Parallel III**" under

Output⇒Cable Setup. Det kan også hjelpe å høyreklikke på prom-chipen (den til høyre) og velge "**Erase**" før man prøver seg på å programmere FPGA-en.



Figur 40. Programmering vellykket

Dersom man skal verifisere programmeringen må man enable readback fra FPGA-en samt å lage en såkalt maskefil. Dette kan gjøre ved å gå inn i prosessvinduet og merke "**Generate Programming File⇒høyreklikk⇒Properties"**. Valgene må være som vist i figuren under.



Figur 41. Enabling av readback for verifisering av programmering

Nå kan man krysse av verify og iMPACT vil lese tilbake konfigurasjonen i kretsen og sammenligne med forventet resultat. Dersom match så er programmeringen verifisert. (Det hender av og til at det

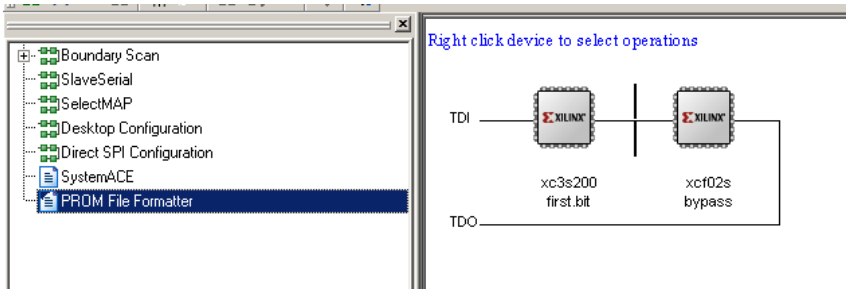
blir feil under denne prosedyren, prøv i så fall de triksene mot feil ved programmering som er beskrevet over)

Vi har nå en metode for å laste konfigurasjonen rett ned i FPGA via JTAG. Under normal drift skal imidlertid FPGA-en hente sine konfigurasjonsdata fra eksternt minne.
I de neste avsnittene skal vi se på hvordan man kan gjøre dette.

Etter å ha lastet ned designet kan dere eksperimentere med det. Vi har en teller som teller oppover til 15. Ved neste flanke vil den telle rundt og starte på 0 igjen. Når LOAD ligger til 1 vil telleren settes synkront til tallet som er på inngangene INP når man gir en positiv flanke på CLK.
Legg merke til at når vi benytter en trykknapp som klokke kan vi få falske klokkepulser av og til. Dette er prell, dvs. metallet i bryteren går mellom av- og påtilstand før den stabiliserer seg i en posisjon.

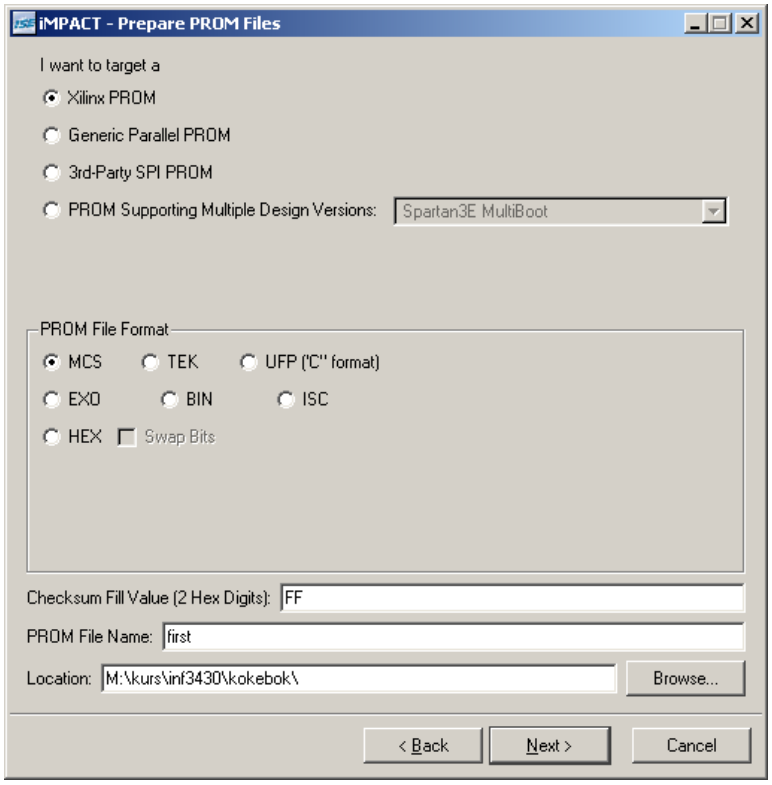
3.8.3 Programmering av konfigurasjonsminnet via JTAG

Nå skal vi programmere konfigurasjonsminnet ved hjelp iMPACT. Mens vi fremdeles er i iMPACT, dobbeltklikk på "PROM File Formatter".

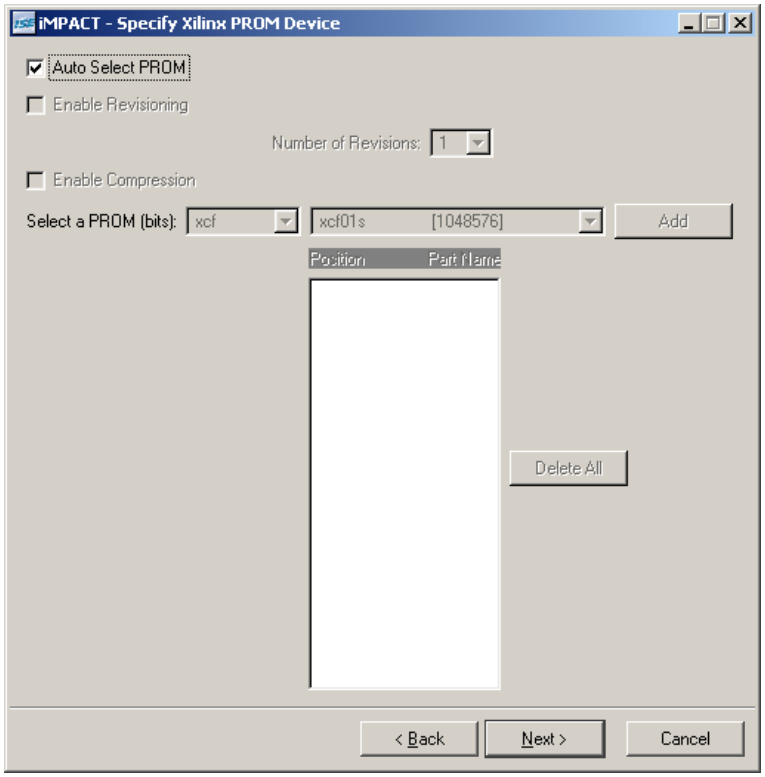


Figur 42

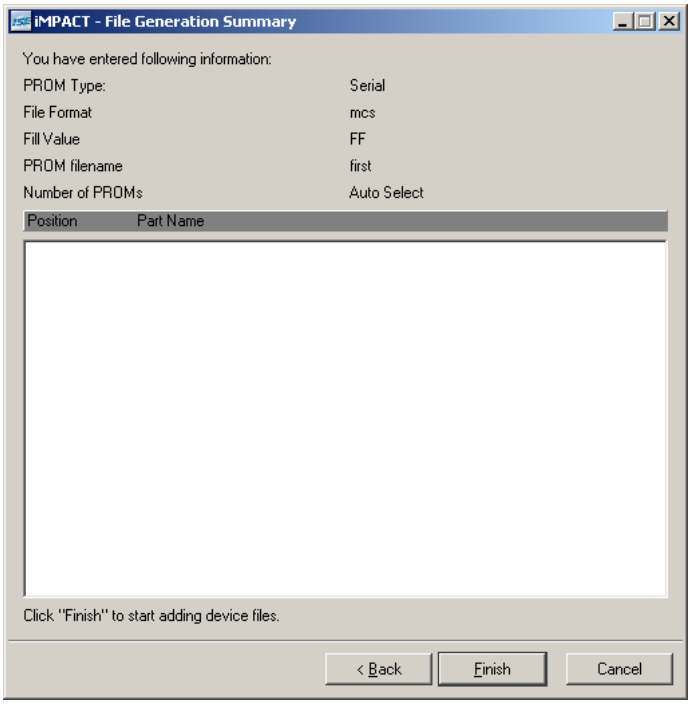
Vi kan ikke benytte .bit filen direkte når skal konfigurere FPGA fra eksternt minne. Det eksterne minnet kan inneholde konfigurasjonen til flere FPGA-er og annet.



Figur 43. Spesifisering av PROM-type(1).

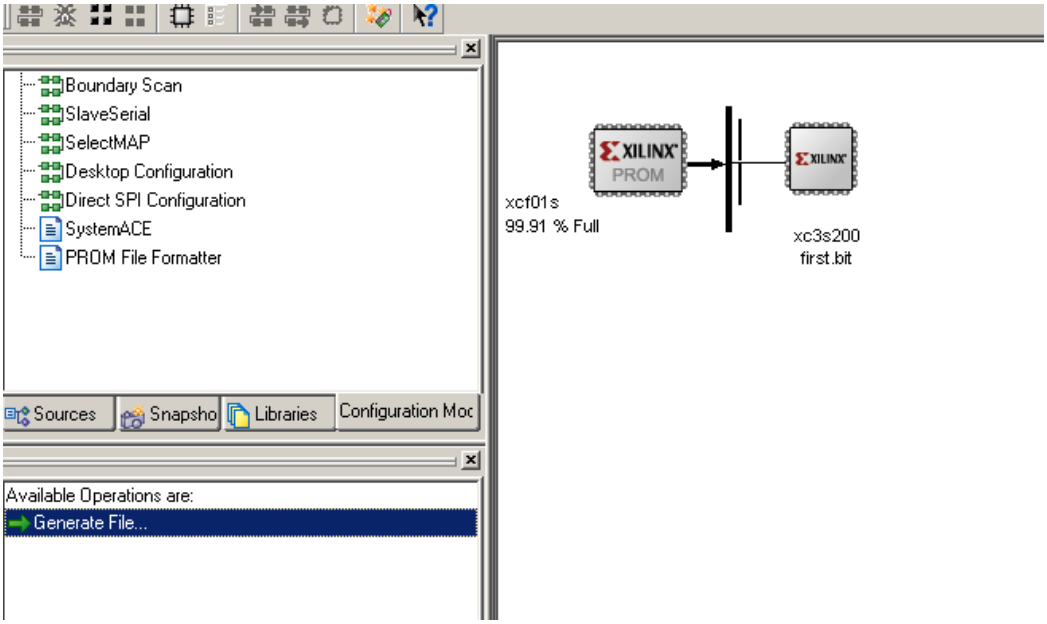


Figur 44. Spesifisering av PROM-type(2).



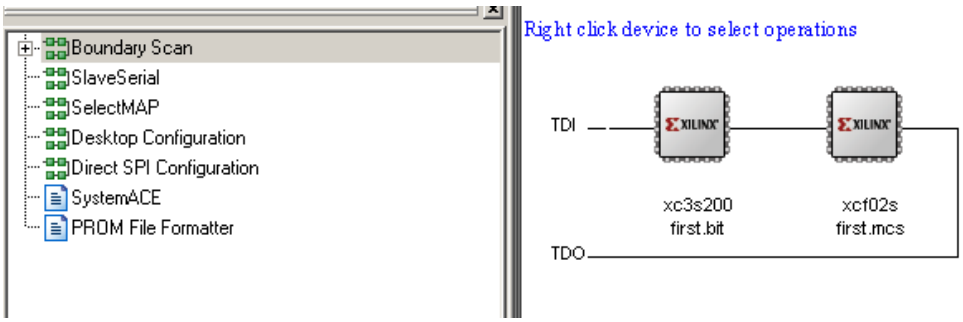
Figur 45. PROM-oppsummering

Etter oppsummeringsbildet, trykk OK, velg first.bit og svar nei på om du vil legge til en fil til, og trykk OK. Da får man følgende skjermbilde, etter å ha klikket på "**Generate File**" :

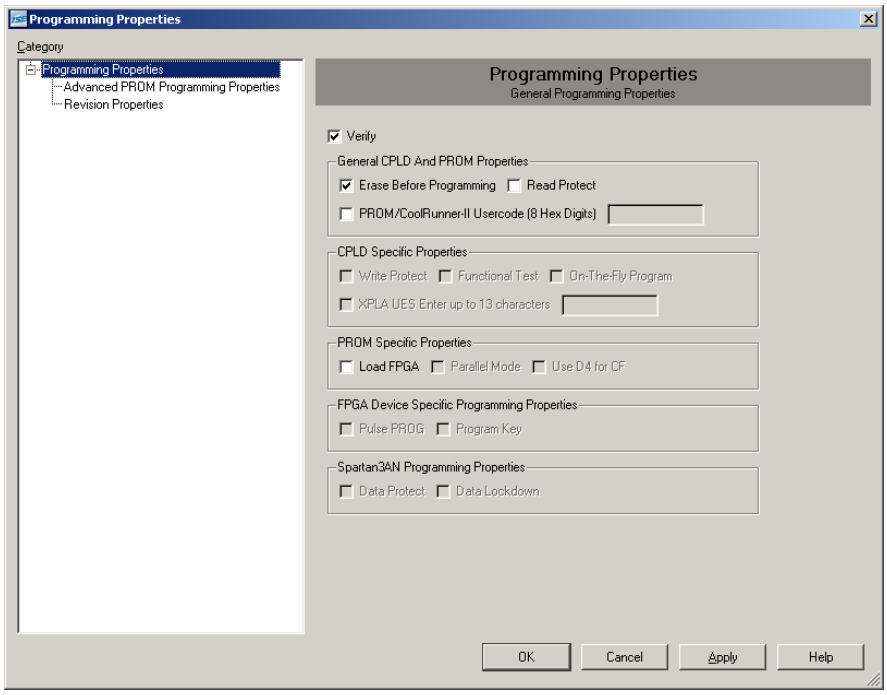


Figur 46

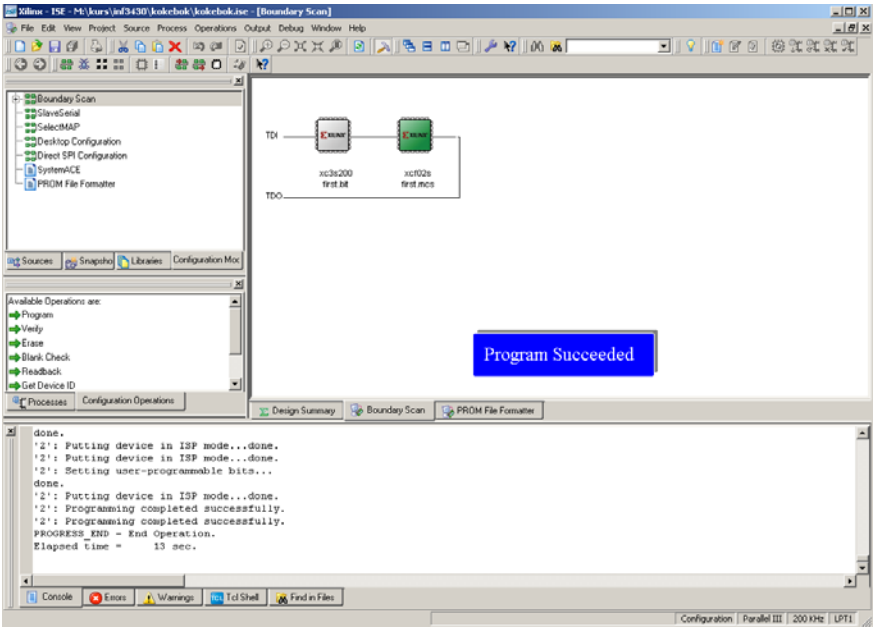
Gå tilbake til "Boundary Scan". Merk **xcf02s⇒høyreklikk⇒ Assign New Configuration File**. Velg first.mcs. Merk **xcf02s⇒høyreklikk⇒Program**.



Figur 47.



Figur 48. Programmering av PROM



Figur 49. Vellykket programmering

Forutsatt at jumperne M0, M1 og M2 er på og JP1 er satt til posisjonen Default vil FPGA-en laste inn konfigurasjonen fra PROM når man slår på strøm eller trykker på PROG-trykknappen. Legg merke til DONE-lysdioden som indikerer at programmering av FPGA er ferdig.

Etter at dere avslutter iMPACT velg å lagre ipf-filen. Ved senere programmeringer kommer dere rett inn i iMPACT slik som vist i Figur 35. Eventuelt endret bit-fil vil automatisk bli tatt med.

4 Editering av VHDL-kode

Dette kapittelet tar for seg nyttig informasjon angående editering av VHDL-kode og andre tekstfiler generelt. Dette er informasjon som bør tas i bruk når dere skal editere filer i laboratorieoppgavene. Test tipsene på en VHDL-kildefil og lær tastatursnarveiene.

4.1 Valg av editor

Teksteditorene som er innebygget i ISE og Modelsim er noe tungvinte å bruke (Dette har bedret seg en del i ISE versjon 9.1). Vi anbefaler at dere bruker en ekstern editor for bedre flyt og kontroll over editeringen. På laben har vi installert Notepad++ (http://notepad-plus.sourceforge.net/uk/site.htm) som er en gratis og kraftig teksteditor. Den har bl.a. mulighet for å markere opp blokker og kommentere ut flere linjer med VHDL-kode. For å få denne til å kjøre automatisk fra ISE går man til Edit→Preferences→ISE General→Editors, velg "Custom" i editor og lim inn `"c:/Program Files/Notepad++/notepad++.exe" -n$2 $1` i "Command line syntax."

4.2 Indentering

Indentering er å rykke inn forskjellige nivåer i koden for å gjøre den mer leselig. Det er ikke nødvendig for at VHDL-koden skal forstås av datamaskinen, men det er helt nødvendig for at andre mennesker skal forstå koden man har skrevet. Eksempler på hvordan dette gjøres kan vi finne masse av i VHDL-boka. I kurset skal all kode som leveres inn være godt indentert.

4.2.1 Eksempel på uindentert og bra indentert kode:

```
COUNTER :
process (RESET,CLK)
begin
if(RESET  = '1') then
COUNT <= "0000";
elsif (CLK'event and CLK = '1') then
if LOAD = '1' then
COUNT <= INP;
else
COUNT <= COUNT + 1;
end if;
end if;
end process COUNTER;
```

Dårlig (ingen) indentering

```
COUNTER :
process (RESET,CLK)
begin
  if(RESET  = '1') then
    COUNT <= "0000";
  elsif (CLK'event and CLK = '1') then
    if LOAD = '1' then
      COUNT <= INP;
    else
      COUNT <= COUNT + 1;
    end if;
  end if;
end process COUNTER;
```

God indentering

4.2.2 Tab/space for indentering

Hvis man bruker tab-tasten for innrykk av programlinjer setter man i utgangspunktet kun inn en tab-kode i dokumentet, ikke et visst antall mellomrom (som man vil gjøre hvis man trykker på mellomromstasten/space). Det blir dermed opp til de forskjellige editorene hvordan programmet vises. Bruk av tab har sine fordeler og ulemper, men det som er sikkert er at det fort blir seendes rart ut i andre editorer hvis et program er skrevet med tab og space om hverandre. *Derfor er det veldig viktig at man ikke blander disse formene for indentering*.

I dette kurset har vi valgt å sette standarden til at kun mellomrom/space skal brukes til indentering, og at det skal brukes 2 mellomrom for hvert nivå. Tab –knappen kan settes opp til å sette inn et visst antall space i stedet for tab-tegnet i dokumentet. Slik slipper man å trykke på space to ganger. I Notepad++ kan dere krysse av dette i **settings→preference→misc→"replace by space"**. "**Tab size**" skal da også være innstilt til 2. I Notepad++ kan en enkelt sjekke om tab eller mellomrom har blitt brukt i dokumentet ved å velge **view→"show whitespace and tab"**.

```
26
27    COUNTER :
28    process (RESET,CLK)
29    begin
30        if(RESET = '1') then
31    ----->----->COUNT <= "0000";
32        elsif (CLK'event and CLK = '1') then
33            -- Synkron reset
34    ----->if LOAD = '1' then
35                COUNT <= INP;
36            else
37    ----->----->COUNT <= COUNT + 1;
38            end if;
39        end if;
40    end process COUNTER;
41
```

Figure 50: Her ser man at det har blitt brukt en blanding av space og tab (pilene markerer tab og prikkene markerer space). Dette bør unngås.

4.3 Kommentarer i koden og variabelnavn

Bruk av kommentarer er viktig for å gjøre koden mer lesbar. Dette er viktig med tanke på samarbeidsprosjekter og gjenbruk. Det er ikke alltid en husker hva ett år gammel kode gjør, selv om man har skrevet den selv. Da kan noen kommentarer være til hjelp.

Bruk av kommentarer er dog ikke en unnskyldning for å lage kryptiske variabel-/signalnavn. Prøv å gi beskrivende navn til inn-/utsignaler og andre elementer i koden. Veldig korte navn bør bare brukes når skopet er kort, som f.eks inne i en enkel process. Hvis koden er selvforklarende med gode variabelnavn trengs ikke kommentarer overalt.

4.4 Bruk av tastatur

Det anbefales på det sterkeste at man setter seg inn i bruk av tastatur og snarveisknapper for editering. Det vil føre til at man editerer raskere, lettere og mer ergonomisk. Bruk av mus for å kopiere og lime inn en linje er tungvint. Notepad++ følger standard snarveisknapper for editering i windows, så det er ikke bortkastet tid å lære disse i alle tilfelle.

4.4.1 Taster og kombinasjoner en må kunne

Taster:

- **home**, **end**, **page up/down**
- **insert**, **delete**
- **tab**

Kombinasjoner:

- **ctrl + piltaster/home/end** for raskere navigering
- **shift + piltaster** for å markere tekst (**ctrl** og **home/end** fungerer også her)
- **ctrl + x/c/v** for cut/copy/paste
- **ctrl + s** for hurtiglagring av dokumentet

For eksempel hvis en vil kopiere en linje med tastaturet kan en først gå til starten av linjen (**home**), for så å merke linjen ved å trykke **shift+pil ned**. Deretter trykker en **ctrl+c** for å kopiere den. Gå så til dit den skal kopieres, og trykk **ctrl+v**.

4.4.2 Spesielle triks for Notepad++

- Hvis man vil kommentere vekk et antall linjer, kan man først merke dem opp og deretter trykke **ctrl+q**. Det samme for å fjerne kommentarene igjen.
- Hvis man vil merke opp kolonner med tekst kan man holde **alt** nede mens man merker opp med shift og piltastene.
- **Ctrl+mushjulet** zoomer inn/ut i teksten.