KRETSKONSTRUKTION MED VHDL

Kurskompendium

ALEXANDER CRAYVENN
SHAWN FARINAM

Projektarbete
Högskoleingenjörsprogrammet för elektroteknik
CHALMERS TEKNISKA HÖGSKOLA
Handledare: Manne Stenberg
Institutionen för signaler och system

Göteborg 2010

INNEHÅLLSFÖRTECKNING

FÖRORD

Detta kompendium är ämnat för studenter inom data- och elektroteknikprogrammet men också för ingenjörer med intresse för konstruktion av integrerade kretsar via ett modernt, hårdvarubeskrivande språk. Kompendiet Kretskonstruktion med VHDL är strukturerat på sådan vis att läsaren kan studera materialet i en kontinuerlig följd, från början till slut, utan att behöva hoppa mellan kapitel och avsnitt. På detta vis erhålls en gradvis kunskapsutveckling där varje nytt kapitel bygger på idéer, exempel och verktyg som introduceras i kapitlen dessförinnan. Kompendiet förklarar grundläggande utvecklingstekniker för logiska kretssystem, dess syntes samt hur de implementeras. Fundamentala lösningar illustreras genom exempel som ger läsaren en lätt överskådlighet av systemens grundläggande byggblock samt hur dessa definieras i ett hårdvarubeskrivande språk. Vi belyser även konstruktionens väg från syntax, syntes till implementering i krets via ett modernt och kraftfullt CAD verktyg.

Målet är att ge läsaren en grundläggande förståelse i VHDL språkets tillämpning i de teknologier som används vid hårdvarukonstruktion av digitala elektroniksystem mha. moderna CAD-verktyg samt hur dessa lösningar har en praktisk betydelse i industrin.

1 INLEDNING

1.1 HISTORIA

Utvecklingen av VHDL påbörjades redan 1981 av det amerikanska försvars-departementet som ett svar på dåvarande livscykelkris för elektronisk hårdvara. Produktionskostnaden för hårdvara nådde en kritisk punkt då tekniken bakom utvecklingsprocessen blev snabbt föråldrad och funktioner för de komponenter som utgjorde ett större system hade vag dokumentation med en individuell verifikationsprocess som översvämmade marknaden med otaligt många simuleringsspråk och verktyg. Behovet av ett standardiserat språk med omfattande beskrivnings och utvecklingsmöjligheter var stort. Men detta var inte nog, krav ställdes att detta nya språk skulle bli funktionsmässigt teknologioberoende, som fungerade likadant oavsett simulering eller konstruktionsmetodik. VHDL har sedan dess utvecklats under kontroll av IEEE som lade grund för den första standardiserade versionen VHDL-87. Men som all standard från IEEE, genomgår språket ständiga småförändringar, främst influerat av användare i näringslivet, därmed ledde detta till en reviderad version VHDL-93 som numera är den standard som stöds av de flesta kommersiella utvecklingsverktyg.

1.2 VAD ÄR VHDL?

VHDL är ett hårdvarubeskrivande språk där förkortningen VHDL kommer från **V**HSIC **H**ardware **D**escription **L**anguage. Akronymen VHSIC står för **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit. Språket kan beskriva ett beteende eller strukturell uppbyggnad av elektroniska system och är främst känt för sin anpassning inom beskrivning av digital elektronikkonstruktion för ASIC- och FPGA-kretsar. VHDL är som tidigare nämnt en internationell standard som regleras av IEEE. En av huvudorsakerna till språkets snabba spridning inom hårdvarudesign är att språket inte är proprietärt (dvs språket ägs inte av någon). VHDL är ett högnivå-, teknologioberoende-, hårvarubeskrivande- språk, som inte är bundet till viss implementationsteknologi eller simulator som har tendensen att begränsa ingenjörens kreativitet. Tvärtom ger detta språk utvecklaren möjligheten att angripa ett problem från flera abstraktionsnivåer samt friheten att välja varierande tekniska utvecklingsmetoder samtidigt som man håller sig inom ramarna till ett enda språk. Då VHDL är teknologioberoende kan en kod flyttas mellan olika kommersiella utvecklingsverktyg utan att någon ändring behöver göras i koden. Stora delar av språket är syntetiserbara där funktioner bestående av miljoner grindar kan implementeras i en krets inom loppet av några minuter. VHDL möjliggör även återanvändning (instansiering) av färdiga komponenter vilket underlättar och snabbar upp en utvecklingsprocess. Ur ett ekonomiskt perspektiv är fördelarna många och tack vare att priserna sjunkit de senaste åren är denna teknologi inte bara tillgänglig för stora företag, men också för mindre bolag som i huvudsak bedriver sin verksamhet inom hårdvarukonstruktion.

2 GRUNDLÄGGANDE EGENSKAPER

2.1 SPRÅKABSTRAKTIONER

VHDL kan användas för att beskriva hårdvara på många abstraktionsnivåer. Detta har effekten att ett komplext systems detaljrikedom göms undan ju högre abstraktionsnivå man har. Med hänsyn till applikationerna av språket för en FPGA/ASIC krets kan det vara till hjälp om man kan identifiera och förstå tre nivåer av hierarkin (abstraktionen).

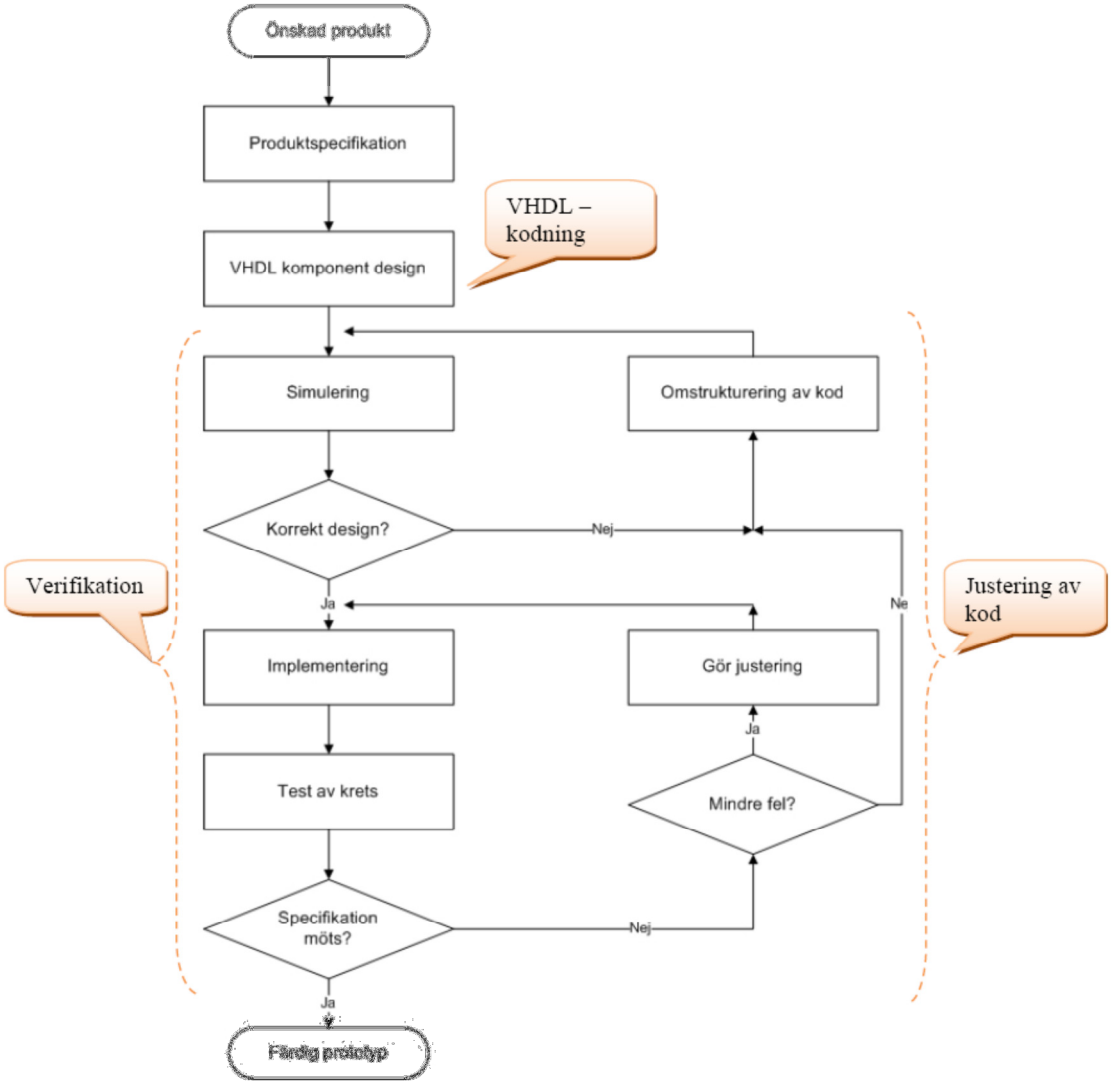
Algoritm (funktionell nivå) består i huvudsak av instruktioner för en uppgift som exekveras sekventiellt. Algoritmen har oftast ingen uppgift om fördröjningar eller klocka och används mest vid simulering och inte syntes.

RTL (Register Transfer Level) är den nivå där man oftast beskriver sina komponenter (t.ex. synkrona, asynkrona nät, register, operatorer). På RTL-nivån definieras kretsens beteende som signaler mellan register (t.ex. D vippa) och logiska operatorer samt hur dessa verkar på signalen.

Grindnivån beskrivs oftast som ett nätverk av grindar (grindnät) eller boolesk algebra.

I dagsläget används RTL-nivån för de flesta beskrivningar. Grindnivån är inte så vanlig och lämpar sig ej för större, komplexa system. Vi strävar efter att beskriva hårdvaran mera efter **vad** denna gör och **inte hur** funktionen utförs. Den funktionella språkabstraktionen är idag inte kompatibel med dagens syntesverktyg. Men i framtiden ser man en ändring av abstraktionsnivåers användning. Utveckling av syntesverktygen går mot högre abstraktionsnivåer och kommer att leda till att dagens norm (RTL) utvecklas mot en algoritmisk beskrivning.

I Figur 2.1 illustreras ett förenklat utvecklingsflödesdiagram som kan vara användbart vid konstruktion av digitala system.



Figur 2.1 Flödesdiagram vid systemutveckling. Det är inte alltid VHDL - kodningen utan verifikationsprocessen som är den mest krävande delen av systemutvecklingen.

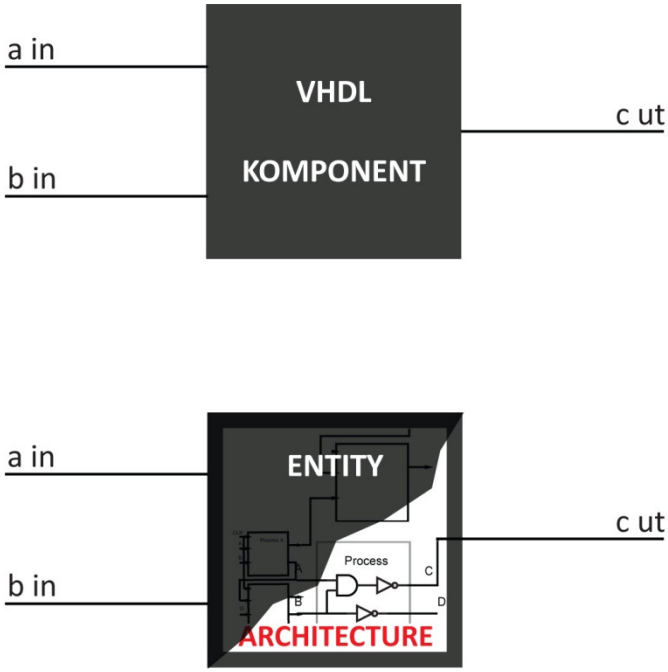
2.2 KOMPONENTBESKRIVNING

Komponenter är ett grundläggande begrepp i VHDL, de är de delar som ett system byggs upp av. Dessa komponenter har den viktiga egenskap att de kan återanvändas, genom att skapa komponentbibliotek där dessa sparas för att därefter möjliggöra instansiering i andra konstruktioner. Notera att en komponent kan bestå av flera andra komponenter som t.ex. CPU, styrkretsar mm.

Samarbetet mellan komponenter i ett större system kan vara svårt att beskriva på beteendenivå i ett högnivåspråk. Komponenterna i sig kan vara nog komplicerade och hur dessa skall samarbeta är språkmässigt ännu svårare att entydigt beskriva. Det är här

strukturella beskrivningen i VHDL kommer till användning. Strukturbeskrivning (Komponentinstansiering) är liktydigt med att koppla samman komponenter.

Varje komponent som beskrivs i VHDL är uppdelad i två huvudbeståndsdelar. En **entitet** (eng. *entity*) deklaration som representerar ett yttre gränssnitt mellan komponentens omgivning och dess portar. Den andra delen är **arkitekturen** (eng. *architecture*) som representerar komponentens interna beskrivning dvs. dess beteende eller struktur.



Figur 2.3 En komponent beskriven i VHDL har en entity (yttre gränssnitt) och architecture (inre struktur/beteende).

Äntligen är det dags att skriva VHDL - kod. På följande sida beskrivs syntaxen för en **entity** följt av två exempel på enkla grindar för att ge en inblick i språkets strukturella uppbyggnad. Som vi vet är syntaxen i ett programmeringsspråk väldigt viktig. Minsta teckenfel i texten medför att koden ej kan kompileras.

En entitet dvs kretsens gränssnitt mot omvärlden beskrivs enligt följande:

```
entity <komponent_identifierare> is
    Port(  <signal_identifierare>: [mode] {data_typ};
           <signal_identifierare>: [mode] {data_typ});
end <komponent_identifierare>;
```

Studerar vi syntaxen till **entity**, så märker man genast att vissa ord som **indentifierare, mode** och **data_typ** dyker upp i texten mer än en gång. Om vi börjar från den första raden och jobbar oss nedåt, så talar den om att **entity** anknyts till ett namn (=**identifierare**). Varje komponent har ett eget namn som t.ex: counter, adder, mux osv. Entitetens in och ut signaler kommunicerar med omvärlden via portar, som man naturligtvis finner på rad två inom parenteserna till ordet **Port**. Här bär varje signal på ett distinkt namn (clk, reset, puch_button) följt av vilken riktning signalen har i förhållande till komponenten och slutligen vilka värden denna signal kan anta. Kommandot **end** i sista raden markerar ett avslut/begränsning av enheten.

Indentifierare

Identifierare används för att namnge saker i en VHDL-modell. Det är en god vana att använda sig av namn som antyder syftet med objektet i fråga. Dock finns det några regler som styr över hur indentifierare kan skrivas:

- Kan endast bestå av bokstäver (’a’ – ’z’ gemener och versaler), decimala siffror (’0’-’9’) och understreck (’_’);
- Måste börja med en bokstav;
- Kan inte avslutas med understreck;
- Kan inte inkludera två efterföljande understreck;

Några exempel på giltiga identifierarnamn är:

```
A   Y1     counter     tryck_knapp     generera_clk_puls
```

Exempel på otillåtna identifierar namn:

```
värde          -- innehåller ett otillåtet tecken
8bit_adderare  -- börjar inte med en bokstav
_Ain           -- börjar med ett understreck
Aut_           -- avslutas med ett understreck
reset__knapp   -- två efterföljande understreck
```

Mode

Mode anger riktningen på data som passerar genom modulen. Dessa är följande:

```
in      -- signalen går enbart in till komponenten som
        -- drivs av någon annan komponent. Skrivs på höger
        -- sida om en tilldelning: r <= c OR inport;

out     -- signalen går enbart ut från komponenten.
        -- Utsignalens värde kan ej läsas i komponenten.
        -- Skrivs på vänster sida om tilldelningsoperatorn:
        -- outport <= a AND b;
```

```
buffer -- signalen går enbart ut från komponenten, men
       -- dess värde är läsbart i komponenten.
       -- kan skrivas på båda sidor om tilldelningen:
       -- buffer_port <= a AND b;
       -- r <= buffer_port OR b;

inout  -- signalen kan gå i båda riktningar. Dvs. signalen
       -- kan läsas eller skrivas av komponenten. Används
       -- vid bidirektionella bussar.
```

data_typ

Indikerar vilken typ av data objektet innehåller. Det finns många datatyper som har support i VHDL men inte många är syntetiserbara. Vi nämner bara ett par av dessa som vi kommer att arbeta med:

```
bit       -- antar värde '1' eller '0';

boolean   -- 'true' eller 'false';

integer   -- kan anta värden mellan -2147483648 till
             2147483647

std_logic -- är den typ vi mest kommer använda då denna
          -- kan anta följande värden:
```

'U' – Uninitialized,	'X' – Forcing Unknown,
'0' – Forcing 0,	**'1' – Forcing 1,**
'Z' – High Impedance,	'W' – Weak Unknown
'L' – Weak 0	'H' – Weak '1'
'-' – Don't care	*** 0, 1 och Z är syntetiserbara**

Observera att språket **inte** skiljer mellan stora och små bokstäver, därmed motsvarar `klocka` och `KLOCKA` samma identifierare. Men vid användning av understreck blir det signifikant då t.ex. `clk_40Hz` och `clk40Hz` indikerar två helt skilda identifierare.

Två efterföljande bindestreck '--' markerar resterande rad som kommentar (ignoreras av syntesverktyg).

EXEMPEL

Följande koder beskriver en två ingångars AND och OR grind:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

Anrop av bibliotek (inbyggda funktioner).

```
entity AND_GRIND is
    Port (a: in STD_LOGIC;
          b   : in STD_LOGIC;
          c   : out STD_LOGIC);
end AND_GRIND;
```

Entitydeklaration. Hur modulens in resp. utgångar förhåller sig till omgivningen.

```
architecture struct of AND_GRIND is

begin
    c <= a AND b;
end struct;
```

Architecturedeklaration. Här förklaras komponentens funktion.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity OR_GATE is
    Port (a: in STD_LOGIC;
          b   : in STD_LOGIC;
          c   : out STD_LOGIC);
end OR_GATE;

architecture struct of OR_GATE
begin
    c <= a OR b;

end struct;
```

data typ

mode
(=signalens riktning)

signaltilldelning

logisk operator

Samtliga grundläggande logiska operatorer (från digitalteknikens värld) är fördefinierade i VHDL, dvs: **NOT, AND, OR, NAND, NOR, XOR och XNOR**.

Innan man skriver sin VHDL kod är det en god vana att alltid ta med ett väldefinierat filhuvud. Ett filhuvud har ingen inverkan på en komponents syntes och implementering, men är däremot viktig för konstruktören, då denna innehåller betydelsefull information om komponenten i fråga. Nedan visas ett exempel på både filhuvud och VHDL - kod för en NOR grind.

```
----------------------------------------------
-- Company:           Company X
-- Engineer:          Alexander Scott Crayvenn
-- Create Date:       19:38:20 03/30/2009
-- Design Name:       NOR.vhd
-- Module Name:       nor - structure
-- Project Name:      Exempel
-- Target Devices:    Cyclone II

-- Tool versions:     Quartus II, ModelSim PE 6.5
-- Description:       Synthesizable model for
--                    an nor gate
-- Errors:            None
-- Additional Comments:
----------------------------------------------
```

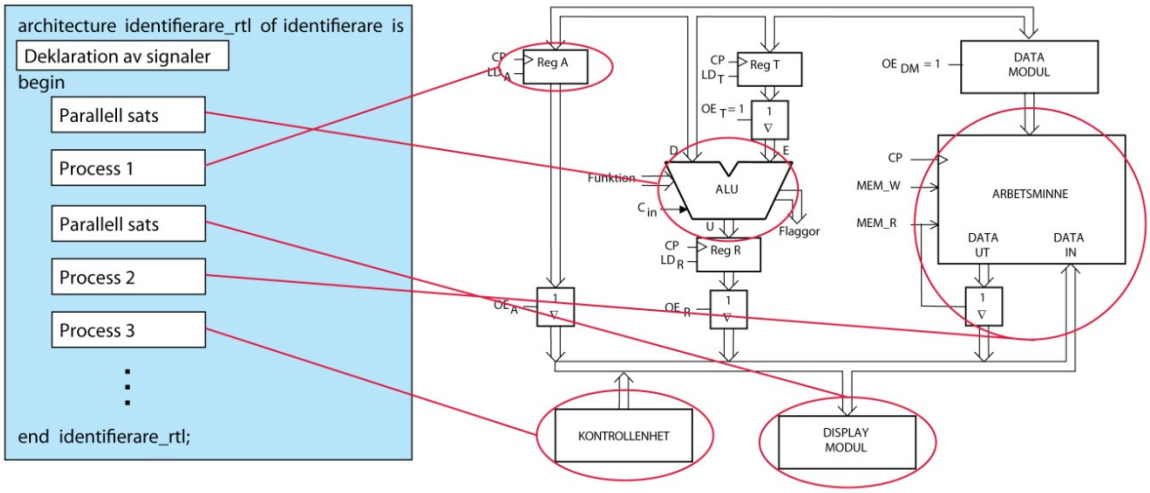
Filhuvud

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity NOR_GRIND is
    Port (a: in STD_LOGIC;
          b   : in STD_LOGIC;
          c   : out STD_LOGIC);
end NOR_GRIND;

architecture structure of NOR_GRIND is
signal temp: STD_LOGIC;
begin
    temp <= a OR b;
    c <= not temp;
end structure;
```

Notera. I denna lösning skapas en intern signal **temp** som först tilldelas OR värdet av insignalerna för att därefter invertera signalen som tilldelas c (=utsignalen).
Men som nämndes tidigare så är operatorn NOR redan definierad i VHDL, så ett alternativt sätt att skriva är helt enkelt: c <= a NOR b;

2.3 EXEKVERING

VHDL-kod kan antingen beskrivas som parallell (concurrent) eller sekventiell. Det är viktigt att kunna särskilja dessa två begrepp för att bättre förstå de sammanhang i vilka de verkar samt hur de berör kodens exekvering. Likt hårdvara är VHDL parallell i sin natur (se figur 2.3). Endast instruktioner som placeras i **processer** eller **funktioner** är **sekventiella**. I ett typiskt programmeringsspråk som C, C++, exekveras instruktionerna i en specifik följd, som bestäms av instruktionernas ordning i källfilen, lika så i processer i VHDL. Men även om exekveringen sker sekventiellt inom dessa processer, så exekveras kodblocken som en helhet, parallellt med andra processer och funktioner. I en VHDL arkitektur finns ingen specificerad ordning för varje instruktion, därmed fastställs ordningen på exekvering av de händelser som inverkar på en viss signal.



Figur 2.3 Visar en enkel dataväg med tillhörande komponenter. I verklig hårdvara körs dessa komponenter parallellt med andra moduler i kretsen, därför måste VHDL kunna beskriva denna parallellitet. Den vänstra sidan av bilden förklarar hur en architecture av en modul kan vara uppbyggd då denna innehåller ett flertal sammanlänkade komponenter. De signaler som deklareras används för kommunikation mellan dessa komponenter och övriga processer i koden. Notera att inom processerna sker exekveringen sekventiellt, men processen som en helhet exekveras parallellt med övriga parallella satser.

3 PARALLELL VHDL

I detta kapitel kommer vi endast beröra parallell (concurrent) kod, dvs. vi kommer att studera de instruktioner som finns utanför procedurer, processer eller funktioner. Dessa beskrivs med hjälp av:
WHEN-, WITH-, GENERATE- statements samt **Operatorer**.

3.1 SIGNALTILLDELNING

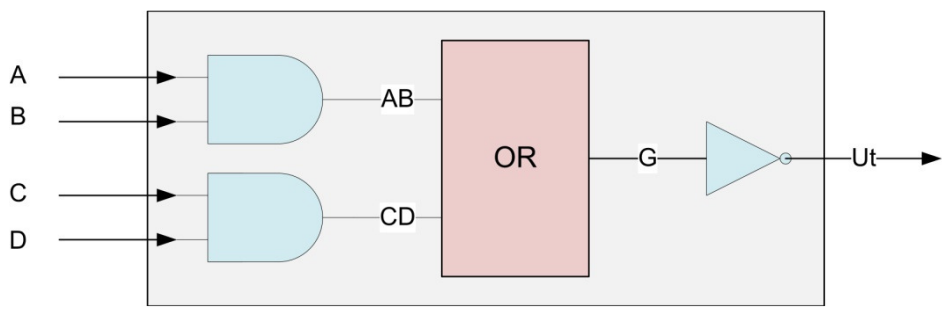
En viktig del av VHDL-syntaxen är signaltilldelningsoperatorn. En signaltilldelning beskriver hur datavärdet överförs från en signal, på höger sida, till en signal på den vänstra sidan om operatorn '<='. Den första signaltilldelningen i exemplet NOR_GRIND (s.12) talar om att data som kommer från insignalerna a och b flyter igenom en OR grind som fastställer värdet på signalen temp (till vänster om operatorn). Tilldelningen i raden efter ger utsignalen c det inverterade värdet av signalen temp.

Man kan se signaler som ledningar som har till uppgift att sammanbinda flera komponenter med varandra eller som verkar inom en komponent mellan sammansättningar av register mm. En signaldeklaration är nödvändig för att skapa en signal. Denna skrivs efter **architecture** men före första **begin** instruktionen (mer om detta sedan). En typisk signaldeklaration kan se ut på följande vis:

```
signal resultat: STD_LOGIC_VECTOR (8 downto 0);
```

Ovanstående rad kan förklaras enligt följande:
Att en signal deklareras syns genom att raden börjar med det reserverade ordet `signal` (i blått). Denna signal (ledning) skall ha ett namn, så vi valde `resultat`. Förutom namnet skall signalens datatyp anges (vad för värden denna kan anta). Det är tydligt att denna signal är en 9 bitars vektor av typen std_logic. Bit 8 till vänster om `downto` markerar den mest signifikanta biten och då drar man slutsatsen att bit 0 till höger representerar den minst signifikanta biten i vektorn.

3.2 INTERNA SIGNALER



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

Entity AOI is
    port ( A, B, C, D : in STD_LOGIC;
           Ut         : out STD_LOGIC);
End AOI;

architecture rtl of AOI is
    signal AB, CD, G : STD_LOGIC;
begin
    AB <= A and B after 3 ns;
    CD <= C and D after 3 ns;
    G  <= AB or CD after 3 ns;
    Ut <= not G after 1 ns;
End rtl;
```

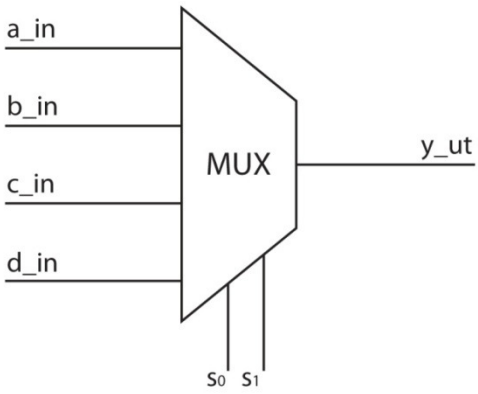
Arkitekturen till kretsen ovan har tre signaler med namnen: AB, CD och G av datatypen STD_LOGIC. Signaler kan ses som portar då dessa läses och tilldelas på samma vis. Signaltilldelningarna innanför ’architecture’ är parallella satser (*eng. concurrent*) som exekveras varje gång en signal på höger sida om tilldelningsoperatorn ändrar värde. Därutav har ordningen på de parallella tilldelningarna ingen betydelse för dess exekvering då potentiellt två eller flera tilldelningar kan exekveras inom samma tidsintervall.

För varje signaltilldelning finns en ingående fördröjning. Uttrycket på höger sida utvärderas varje gång en värdeändring sker som leder till att vänstra sidan om operatorn uppdateras efter en viss fördröjning. En VHDL-kod med fördröjningar (ovan) går att kompilera men dessa utelämnas vid syntes. De är däremot värdefulla vid simulering.

3.3 OPERATORER I PARALLELL KOD

Användning av operatorer (NOT, AND, OR, -, + …) är det mest elementära sättet att skapa parallell VHDL kod. Dessa operatorer kan användas för att skapa kombinatoriska kretsar av alla slag. Dock kommer det visa sig att mer komplexa kretsar beskrivs enklare via sekventiell kodning även om kretsen i sig inte innehåller sekventiell logik.

För att få en bättre förståelse av detta kommer det i exemplet som följer på nästa sida beskrivas en väljare (MUX) genom att endast använda logiska operatorer.



Figur 3.2.1 En 4 till 1 (4:1) väljare (mux)

Figuren ovan visar en 4 ingångars multiplexer med en bit per ingång. Utgången skall vara lika med den ingång som väljs av signalerna s0 och s1. En implementering av denna krets via logiska operatorer kan göras enligt följande exempel:

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

entity MUX is
    Port (    a_in   : in  STD_LOGIC;
              b_in   : in  STD_LOGIC;
              c_in   : in  STD_LOGIC;
              d_in   : in  STD_LOGIC;
              s0, s1 : in  STD_LOGIC;
              y_ut   : out STD_LOGIC );
end MUX;

architecture Structure of MUX is
begin
    y_ut <=(a_in AND NOT s1 AND NOT s0) OR
           (b_in AND NOT s1 AND s0) OR
           (c_in AND s1 AND NOT s0) OR
           (d_in AND s1 AND s0);
end Structure;
```

3.4 WHEN / WITH STATEMENT

Som nämndes i början av kapitel 3 är WHEN och WITH två grundläggande parallella satser i VHDL. I exemplet som följer visas lösningen till föregående exempel (mux) genom att beskriva komponentens beteende just med dessa två satser.

---------- Med WHEN Sats ----------

```
architecture behavioral of MUX is

signal sel : std_logic_vector(1 downto 0);

begin
    sel <= s1 & s0;

    y_ut   <=  a_in when sel = “00” else
               b_in when sel = “01” else
               c_in when sel = “10” else
               d_in;

end behavioral;
```

---------- Med WITH Sats ----------

```
architecture behavioral of MUX is

signal sel : std_logic_vector(1 downto 0);

begin
    sel <= s1 & s0;

    WITH sel SELECT
        y_ut   <=  a_in when “00”,
                   b_in when “01”,
                   c_in when “10”,
                   d_in when others;

end behavioral;
```

3.5 RELATIONSOPERATORER

Likt andra programmeringsspråk har VHDL relationsoperatorer som man kan ha nytta av i många processer.

Operationssymbol	Beskrivning
=	Lika med
/=	Skilt ifrån
<	Mindre än
>	Större än
<=	Mindre eller lika med
>=	Större eller lika med

Observera! *Blanda ej ihop '<=' relationsoperatorn med signaltilldelningsoperatorn, då de skrivs på samma sätt.*

3.6 SHIFT OPERATORER

Shift operatorer kan väl komma till hands då man med **en** rad kod kan beskriva ett skiftregister. Dessa operatorer är:

Operator	Beskrivning
sll	skift åt vänster *(eng. shift left logical)*
srl	logisk skift åt höger *(eng. shift right logical)*
sla	aritmetisk skift vänster *(eng. shift left arithmetic)*
sra	aritmetisk skift höger *(eng. shift right arithmetic)*
rol	rotera vänster *(eng. rotate left)*
ror	rotera höger *(eng. rotate right)*

För bättre förståelse visas exempel för varje operator. Låt oss anta att vi har signalen A som tilldelas vektorn ”10110101”.

```
A sll 2 = ”11010100” -- logisk skift 2 steg åt vänster, resten fylls
                     -- med nollor.
A srl 2 = ”00101101” -- samma som ovan fast med logisk skift höger.
A sla 3 = ”10101111” -- aritmetisk skift vänster 3 steg,
                     -- resterande bitpositioner fylls med värdet av
                     -- den högra biten i strängen.
A sra 2 = ”11101101” -- aritmetisk skift höger 2 steg.
A rol 4 = ”01011011” -- rotera vänster 4 bitar.
A ror 3 = ”10110110” -- rotera höger 3 bitar.
```

4 SEKVENTIELL VHDL

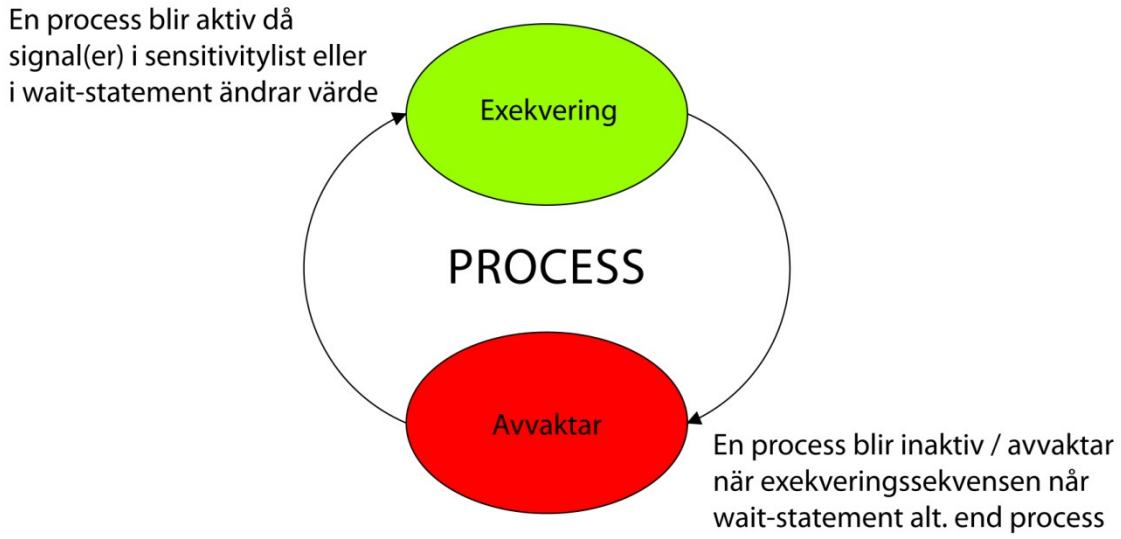
Som vi nämnde i det föregående kapitlet är VHDL kod parallell i sin natur. Processer var däremot de delar av koden som exekveras sekventiellt. Emellertid är dessa block parallella med andra utsagor utanför processen. En viktig aspekt är att sekventiell kod är inte enbart begränsad till sekventiell logik, utan möjliggör även uppbyggnad av kombinatoriska kretsar. I detta kapitel kommer vi reda ut hur processer kan användas för att beskriva komponenter.

4.1 PROCESS

En process är en sekventiell sektion i VHDL kodning. Den tillkännages genom förekomsten av IF, CASE eller LOOP satser. En process skall anges under architecture efter första begin. I processens *sensitivitetslista* ingår de signaler som skall påverka processens exekvering. Varje gång en signal i sensitivitetslistan berörs (ändrar värde) startar processen. Syntaxen för en process visas nedan:

```
[process_namn:] process (sensitivitetslista)
<variabel namn :data_typ {:= initial_värde;}
begin
<sekventiella satser>
end process [process_namn];
```

Om variabler används skall dessa anges i processens deklarationsdel (mellan **process** och **begin**). Initialvärdet för variabeln är ej syntetiserbart och används endast vid simulering. Användning av `process_namn` är valfritt men rekommenderas då det förtydligar kodens syfte och läsbarhet.



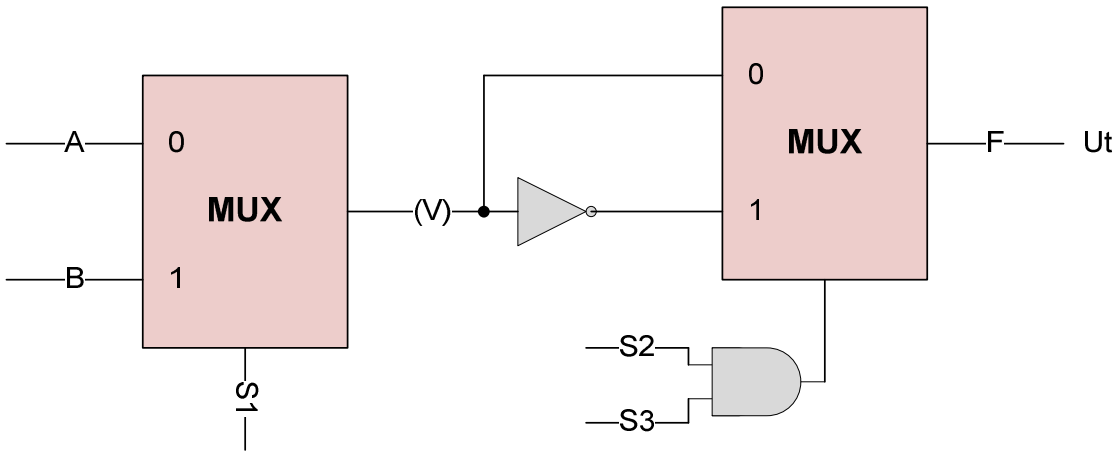
Exempel på aktivering av processer:

```
process(x, y, z)
begin
    Ut <= (x and y) or z;
end process;
-------------------------------------------------
process
begin
    Ut <= (x and y) or z;
wait on x, y, z
end process;
-------------------------------------------------
```

4.2 IF STATEMENT

En **if** statement är en sekventiell sats som exekverar andra sekvens-satser då ett villkor uppfylls. En if statement kan efterföljas av en **else** del som exekveras då villkoret / påståendet under **if** är falskt. Varje if statement skall avslutas med en korresponderande **end if**. Observera att end if är två separata ord – att skriva endif är ett vanligt förekommande fel!

EXEMPEL
Ett if-statement syntetiseras genom skapandet av en multiplexer för de signaler och variabel som tilldelas if satsen. Select signalen till varje multiplexer drivs mha tillståndet av if satsens logiska utfall och dess dataingångar bestäms av uttrycket på höger sida om tilldelningsoperatorn (se figur med tillhörande VHDL-kod nedan).



```
process (S1, S2, S3, A, B)
    variable V : STD_LOGIC;
begin
    if S1 = '0' then
       V:= A;
    else
       V:= B;
    end if;

    if S2 = '1' and S3 = '1' then
       V:= not V;
    end if;

    F <= V;
end process;
```

4.3 VARIABLER kontra SIGNALER

En variabel i VHDL är en temporär minnesallokering som deklareras och användas innanför en process, - till skillnad från signaler kan variabler **ej** användas för kommunikation mellan processer. Den mest påtagliga skillnaden är dock att en signaltilldelning alltid kommer med en fördröjning, medan variabeltilldelningen är omedelbar. På detta sätt kan signaler ha en kommande vågfront av värden och händelser som komma skall.

```
process (A, B, C)
    variable V : STD_LOGIC;
begin
    V := A and B;
    V := V or C;
    UT <= not V;

end process;
```

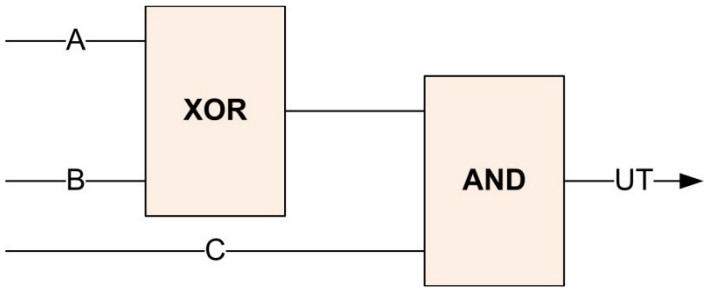
I koden ovan, allokerar variabeldeklarationen (variable V : STD_LOGIC;) en minnesplats för variabeln V av typen STD_LOGIC. Variabeltilldelningen (V :=) skriver in ett nytt värde på den reserverade platsen och gör detta värde omedelbart tillgänglig för nästföljande sekventiella satser.

I vissa fall kan variabler användas istället för signaler medan i andra fall är det inte att rekommendera. För att använda variabler och signaler på ett korrekt sätt skall man hålla följande punkter i minnet:

- Variabler deklareras och användas i en process (Observera att en variabel kan endast användas i den process den deklareras i). Olikt signaler kan variabler ej användas för kommunikation mellan processer.

- Variabler kan inte användas i en sensitivitetslista, ty är det ej möjligt att aktivera en process vid variabeluppdatering.

- En variabeltilldelning saknar fördröjningar och är därmed omedelbar, till skillnad från signaler som alltid har en ’medfödd’ intern fördröjning.



1

```
UT <= (A xor B) and C;
```

2

```
process ( A, B, C )
    variable TEMP: STD_LOGIC;
begin
    TEMP := A xor B;
    UT   <= TEMP and C;
end process;
```

3

```
signal TEMP: STD_LOGIC;
…
process (A, B)
begin
    TEMP <= A xor B;
end process;

process (TEMP, C)
begin
    UT <= TEMP and C;
end process;
```

De tre olika kodsnuttarna beskriver samma logiska krets ovan. Den första koden beskriver en signaltilldelning där parentes används för att bestämma ordningen/prioriteringen på uttryckets exekvering. I VHDL koden som beskrivs i ruta 2, deklareras variabeln ’TEMP’ för mellanlagring av värdet. I det tredje alternativet används en signal för mellanlagring. Konsekvensen av detta kräver två processer i koden.

EXEMPEL

Nu när vi har gått igenom parallell och sekventiell vhdl kodning är det hög tid för ett exempel, där lösning är möjlig med båda metoder. Frågan är om dessa två skrivsätt leder till samma implementerad hårdvara.

Skriv en VHDL – kod (entity och architecture) till nedanstående sanningstabell:

- Entity namn: exempel
- Architecture namn: rtl
- Insignaler: X0, X1
- Utsignal: Y

Signalerna är av typen std_logic.

X0	X1	Y
0	0	1
0	1	0
1	0	1
1	1	1

För ej angivna tillstånd skall Y ha värdet 0.

Lösning 1

```
library ieee;
use ieee.std_logic_1164.all;

entity exempel is
    Port ( X0 : in  STD_LOGIC;
           X1 : in  STD_LOGIC;
           Y  : out STD_LOGIC);
end exempel;

architecture rtl of exempel is
begin
process (X0, X1)
    subtype select_type is std_logic_vector (1 downto 0);
    begin
        case select_type'(X0 & X1) is
            when "00"|"10"|"11" => Y <= '1';
            when others => Y <= '0';
        end case;
end process;
end rtl;
```

Lösning 2

```
library ieee;
use ieee.std_logic_1164.all;

entity exempel is
    Port ( X0 : in  STD_LOGIC;
           X1 : in  STD_LOGIC;
           Y  : out STD_LOGIC);
end exempel;

architecture rtl of exempel is

signal X_value : STD_LOGIC_VECTOR(1 downto 0);

begin
X_value <= (X0 & X1);

Y <= '1' when X_value = "00" or "10" or "11" else
     '0';

end rtl;
```

En viktig tumregel för en konstruktör är att känna till hårdvaran som skapas av motsvarande VHDL kod. Lösning 1 och Lösning 2 kommer inte att resultera i samma hårdvara. Man kan se tydligt att Lösning 2 kommer skapa en MUX (when statement) medan i Lösning 1 bygger på en process som inte är beroende av en klocka där case satsen kommer resultera i att minneselement jämförs innan tilldelning till utsignalen. Den första koden är opraktisk, dels för att konstruktören ej vet vad koden skapar (syntesen beror också i detta fall av vilket verktyg som används) samt att denna tar upp mer utrymme i FPGA'n.

5 SYNKRON och ASYNKRON kodning

I digitalteknikens grunder har vi lärt oss att synkrona processer koordineras i en tidsaxel (är beroende av en klocka). Medan det motsatta gäller för asynkrona nät.

Vid kodning av synkrona processer, är det en god vana att ta med en asynkron reset. Skillnaden mellan synkron och asynkron reset visas i kodexemplen nedan:

Asynkron reset

```
process(RST,CLK)
begin
    if (RST='1') then  -- högaktiv reset
        counter <= ( others => '0' );

    elsif rising_edge (CLK) then
        counter <= counter + '1' ;

    end if;
end process;
```

Synkron reset

```
process (CLK)
begin
  if rising_edge(CLK) then
    if RST='1' Then -- aktiv hög reset
      counter <= ( others => '0' );
    else
      counter <= counter + 1;
    end if; -- Reset
  end if; -- CLK
end process;
```

Notera att asynkron reset beskrivs före klockan (CLK) i if-satsen medan vid synkron reset skrivs denna efter klockan. Förklaringen är enkel. Som vi vet sedan kapitel 4, exekveras en if-sats sekventiellt. Detta betyder att påståendet som står först i if-satsen har en större prioritering. Notera även att vid en synkron reset så finns inte RST-signalen med i processens sensitivitetslista.

Andledningen till att vi vill använda asynkron reset är att minneselementen i FPGA kretsen är utrustade med asynkron reset, därmed är det *”gratis”* att använda asynkron reset.

6 GENERICS

Översatt till svenska betyder ‘generic’ allmän, som är ett reserverat ord i VHDL och används vid specifikation av allmänna, statiska parametrar som med modifikation kan anpassas för olika applikationer. Syftet med **generics** är att göra koden mer flexibel, särskilt vid förekomsten av databussar. Denna deklareras under komponentens entity där den ses som en global parameter. Syntaxen för en generic-deklaration visas nedan:

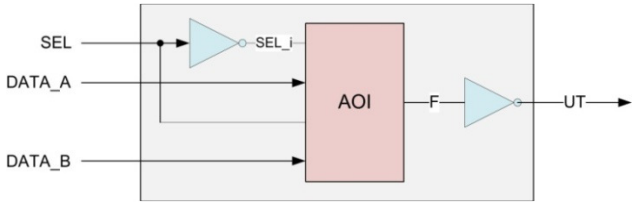
```
GENERIC ([namn] : data_typ := [värde]);
```

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

Entity Comp is
Generic (n: INTEGER := 4);
    port ( DATA_A : in STD_LOGIC_VECTOR(n-1 downto 0);
           DATA_B : in STD_LOGIC_VECTOR(n-1 downto 0);
           L_TH   : out STD_LOGIC;
           EQ     : out STD_LOGIC;
           GR_TH  : out STD_LOGIC);
End Comp;

architecture behv of Comp is
begin
    process (DATA_A, DATA_B)
    begin
        if (DATA_A < DATA_B) then
            L_TH  <= '1';
            EQ    <= '0';
            GR_TH <= '0';

        elsif (DATA_A = DATA_B) then
            L_TH  <= '0';
            EQ    <= '1';
            GR_TH <= '0';

        else
            L_TH  <= '0';
            EQ    <= '0';
            GR_TH <= '1';

        end if;
    end process;
end behv;
```

I exempelkoden ovan används GENERIC för att specificera parametern ’n’ av typen INTEGER vars värde anges till 4, dvs. var än i koden (entity eller architecture) parametern ’n’ står angiven, så kommer dess värde att förknippas med 4.

7 KOMPONENT DEKLARATION/INSTANSIERING

Figuren nedan visar komponenten ’MUX’ som i sin tur består av andra subkomponenter. Koden är strukturellt uppbyggd på en högre abstraktionsnivå. Komponentdeklarationerna (NOT och AOI) i koden skall ha samma motsvarighet som dess entitet med avseende på namn, portar och datatyp. I arkitekturen STRUCTURE görs instansiering av de två subkomponenterna INV och AOI. Komponentnamn efter ’*component*’ är referenser till entiteter som är definierade utanför MUX. Portnamnen i komponentdeklarationen används för identifiering av portar under ’**port map**’ (=namnassociation). Notera att både INV samt AOI komponenten har portar med samma namn. Detta är fullt tillåtet i VHDL.




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

Entity MUX is
    port ( SEL    : in STD_LOGIC;
           DATA_A : in STD_LOGIC;
           DATA_B : in STD_LOGIC;
           UT     : out STD_LOGIC);
End MUX;

architecture structure of MUX is
    component INV
      port(A : in  STD_LOGIC;
           F : out STD_LOGIC );
    end component;

    component AOI
      port(A, B, C, D : in  STD_LOGIC;
           Ut         : out STD_LOGIC );
    end component;

    signal SEL_i, F : STD_LOGIC;

begin
    inst_1: INV
      Port map ( A => SEL,
                 F => SEL_i );

    inst_2: AOI
      Port map ( A  => SEL_i,
                 B  => DATA_A,
                 C  => SEL,
                 D  => DATA_B,
                 Ut => F );

    inst_3: INV
      Port map ( A => F,
                 F => Ut );

End structure;
```

Port map kan antingen beskrivas genom namnassociation eller positionsassociering. Med namnassociation anges komponentens portnamn och de signaler de ansluts till, medan vid positionsassociering är det ordningsföljden av signalerna i port map som bestämmer vilken port de ansluts till.

För enkelhets skull visas ett exempel nedan som klargör skillnaden.


```
Component KOMPONENT
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : out STD_LOGIC );
End component;

Signal DATA_A, DATA_B, SEL : STD_LOGIC;
```

Namn association

```
K1: KOMPONENT port map (A => DATA_A, B =>DATA_B, C =>SEL);
```

Position associering

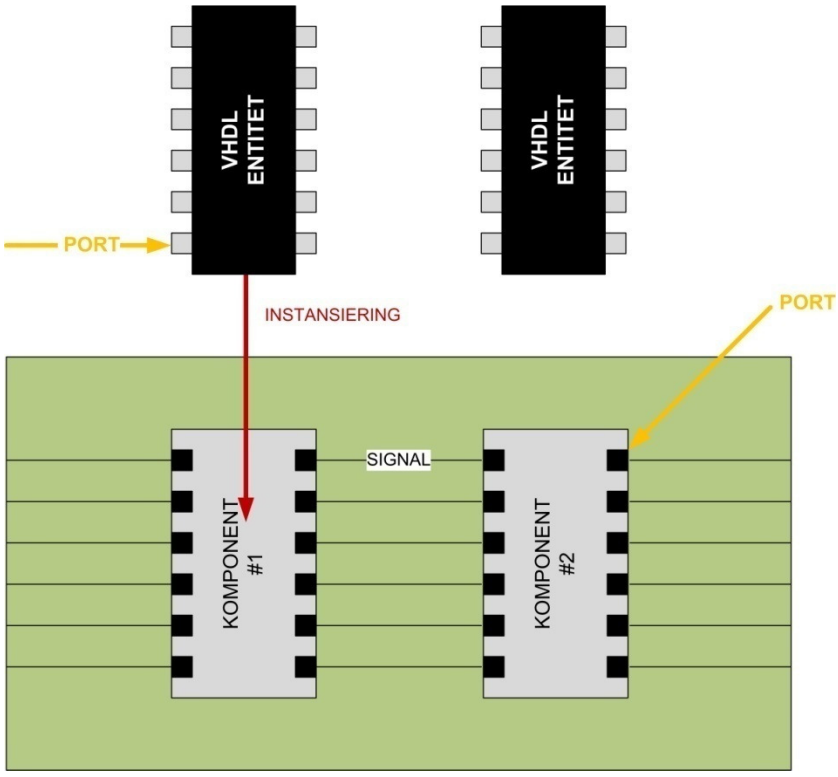
```
K1: KOMPONENT port map (DATA_A, DATA_B, SEL);
```

7.1 DIREKT INSTANSIERING

Med direkt instasiering slipper man krångel med komponentdeklarationer i koden. Dock har denna metod viss nackdel. Entiteten som instansieras skall dessförinnan redan vara skriven och kompilerad, samt att ens design inte är lika flexibel. Om man tänker sig figuren på nästa sida (kretskortanalogin), så motsvarar denna metod en direkt lödning av komponenten på kretskortet vilket försvårar ett eventuellt byte av komponenten i fråga.

I exemplet som följer visas en VHDL kod där direkt instansiering används för att beskriva vår tidigare kända komponent, multiplexern.

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

Entity MUX is
    port ( SEL     : in STD_LOGIC;
           DATA_A  : in STD_LOGIC;
           DATA_B  : in STD_LOGIC;
           UT      : out STD_LOGIC);
End MUX;

architecture STRUCTURE of MUX is
    signal SEL_i, F : STD_LOGIC;
begin
    G1: entity WORK.INV(rtl) port map (A=> SEL,   F=> SEL_i);
    G2: entity WORK.AOI(rtl) port map (A=> SEL_i, B=> DATA_A,
                                       C=> SEL,   D=> DATA_B,
                                       Ut=> F);
    G3: entity WORK.INV(rtl) port map (A=> F,     Ut=> UT);
End STRUCTURE;
```



Att lära sig nya definitioner och ord i ett nytt språk kan vara förvirrande då man har svårt att associera dessa med något verkligt. Men vi har turen på vår sida för VHDL beskriver just något verkligt dvs. hårdvara. På bilden ovan tar vi hjälp av kretskort analogin som illustrerar vad man i själva verket menar med VHDL entitet, port, komponent, signal och instansiering.

8 TILLSTÅNDSMASKINER (*eng. FSM = Finite State Machine*)

8.1 MEALY / MOORE

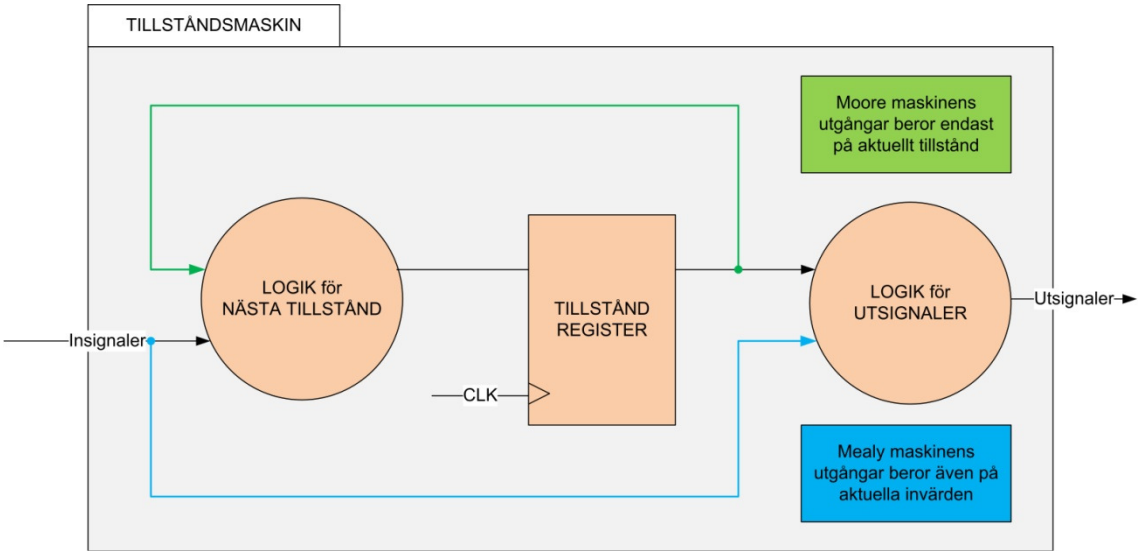
En tillståndsmaskin är en väldigt användbar abstraktion för konstruktion av digitala kretsar. Som det engelska namnet antyder är tillståndsmaskinen digital logik med ett ändligt antal fasta tillstånd. En FSM använder värdet av dess ingångar och aktuella tillstånd för att bestämma värdet på dess utgångar samt nästföljande tillstånd. I detta häfte tar vi endast upp synkrona FSM dvs. där tillståndet ändras på en aktiv klockflank.

Själva hårdvaran (arkitekturen) av en tillståndsmaskin består av ett tillståndsregister, kombinatorisk logik för att bestämma nästa tillstånd samt kombinatorisk logik för utsignalerna (se bild nedan).

En tillståndsmaskin kan i huvudsak beskrivas på två olika sätt, döpta efter dess grundare Moore och Mealy.

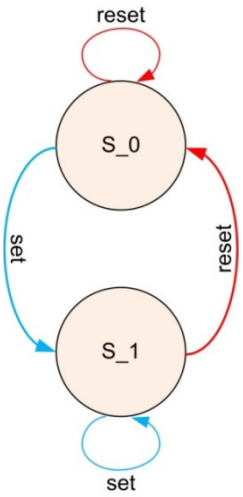
- Utsignalerna av en **Moore** maskin är endast beroende av det aktuella tillstånd som maskinen befinner sig i. Detta leder till att utsignalerna ändrar värde endast när tillståndsmaskinen byter tillstånd.

- Utsignalerna av en **Mealy** maskin är dels beroende av de aktuella värden på ingångarna och dels det aktuella tillståndet. På så vis kan en ändring på inportarna leda till en ändring i utsignalerna utan att invänta nästa tillstånd.



EXEMPEL

En SR-vippa beskriven mha en tillståndsmaskin

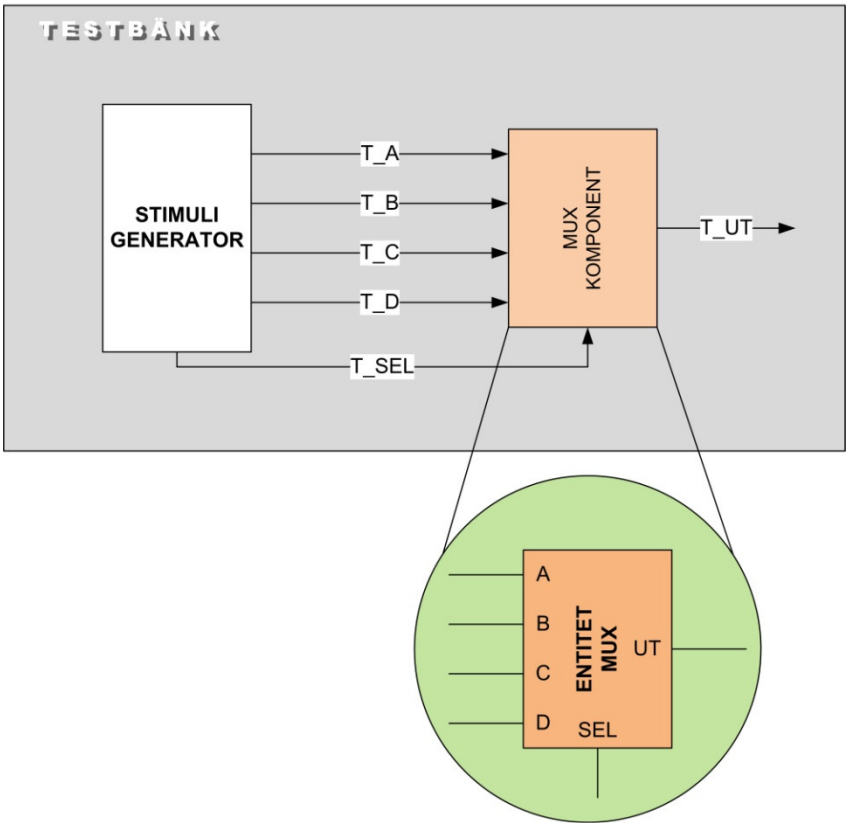


Nedan visas en grundläggande mall som beskriver en tillståndsmaskin i VHDL.

```
ARCHITECTURE Arc_FSM OF FSM IS
 TYPE StTyp IS (S0, S1, S2, S3);
 SIGNAL NState, State: StTyp;

BEGIN
------- Synkron Process ------------
Synk_proc:PROCESS(clk_50, Reset)
        BEGIN
                IF Reset='0' THEN
                        state <= S0;
                ELSIF rising_edge(clk) THEN
                                state <= NState;
                END IF; -- CLK
        END PROCESS Synk_proc;
--- Main Process ----------
  PROCESS (state,....)
        BEGIN
                port/signal <= värde; -- Defoult
                CASE state IS
-----------------------------------------------------
                  WHEN s0 =>
                        port/signal <= värde; -- Moore output
                        IF cond THEN
                          port/signal <= värde; -- Mealy output
                          NState<=S1;
                        ELSE
                           port/signal <= värde; -- Mealy output
                           NState<=S2;
                        END IF;
-----------------------------------------------------
                        WHEN S1 =>
                                . . .
                        WHEN S2 =>
                                . . .
                        WHEN S3 =>
                                . . .
                        WHEN OTHERS =>
                          NState<=S1;
                END CASE;
        END PROCESS;
END Arc_FSM;
```

9 TESTBÄNKAR

VHDL möjliggör inte bara beskrivning av hårdvara och digitala system utan även möjligheten att simulera dessa via testbänkar som genererar in stimuli till designen och analysera resultatet. Denna mångsidighet gör att en kod kan ersätta dyra oscilloskop, signalgeneratorer och specialbyggda komponentinterface. En annan aspekt av dess mångsidighet är då en testbänk skrivs som en .vhd fil kan denna flyttas mellan CAD verktyg. Figur (.) illustrerar en generell uppbyggnad av en testbänk.



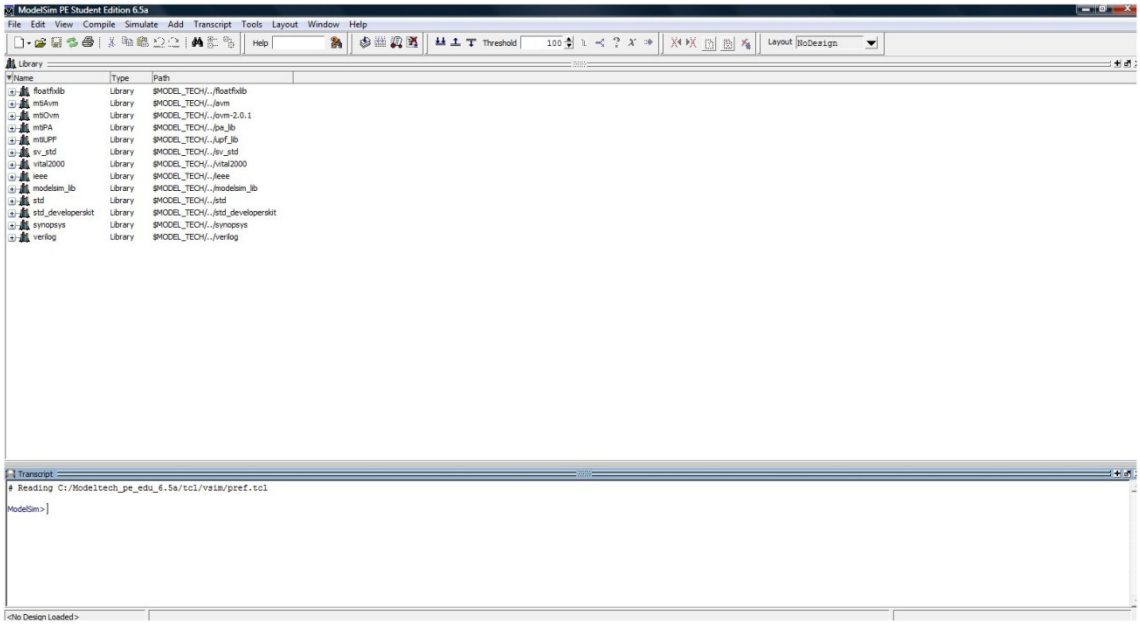
Figur (.). Testbänken genererar insignaler till komponenten som befinner sig under testläge (*eng. uut = unit under test*). För att manuellt bekräfta kretsens funktionalitet behövs ett verktyg som ModelSim (waveform editor). Om man inte har lust att se på ett diagram kan en testbänk skrivas så att den själv märker av fel i utsignalen och rapporterar detta till konstruktören via ett felmeddelande.

10 MODELSIM

ModelSim är ett väldigt användbart verktyg för kompilering och simulering av VHDL kodade komponenter. Denna mjukvara finns tillgänglig för gratis nedladdning under länken http://www.model.com/downloads/default.asp med namnet ModelSim PE Student Edition vilket är en något begränsad version men fullt tillräcklig för våra konstruktioner.

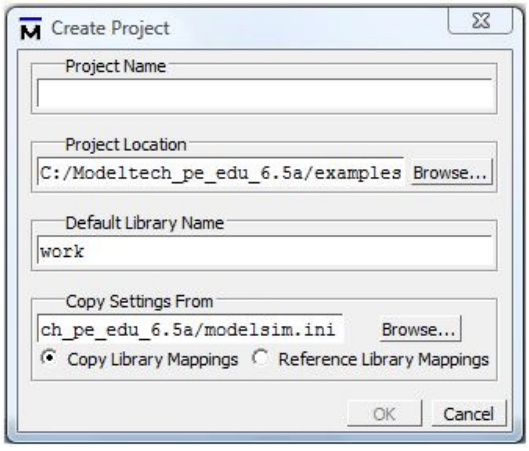
10.1 START AV NYTT PROJEKT

Vi antar att läsaren är bekant med PC och en Windows miljö, därav hoppar vi över alla onödiga grunder och går rakt på sak.

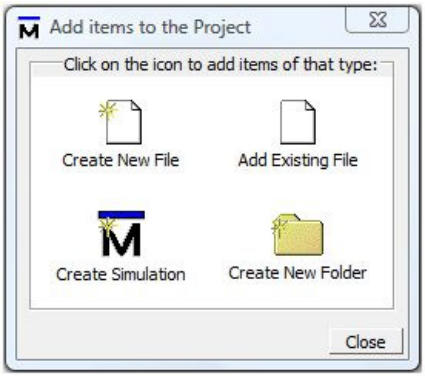


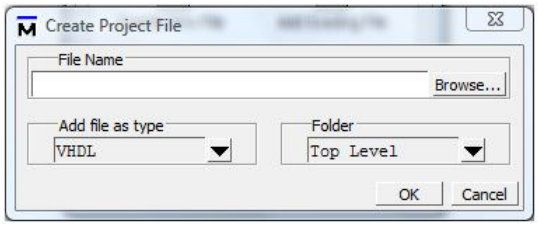
Bilden ovan visar hur ModelSim kan se ut vid uppstart

Det första som bör göras om man vill skapa en VHDL komponent i ModelSim är att starta ett nytt projekt. Men innan man påbörjar med detta är det rekommenderat att skapa en ny mapp där projektet kommer sparas i, då programmet ej gör detta åt en. Man skapar ett nytt projekt under File menyn, genom att välja: File => New => Project.

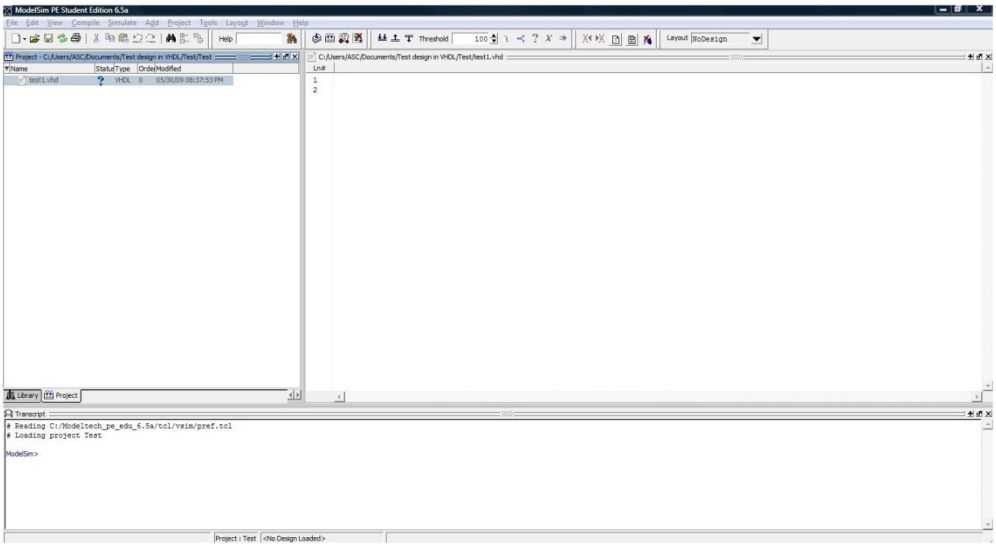


I det nya fönstret som öppnas (till vänster) anger man projektnamnet och dess plats (den nyskapade mappen). Under 'Default Library Name' anges namnet på den under mapp där all kompilerad kod kommer att hamna. Det rekommenderas att låta dessa inställningar vara orörda. När man är färdig trycker man givetvis på 'OK'. Detta öppnar ett nytt fönster (se nästa sida).

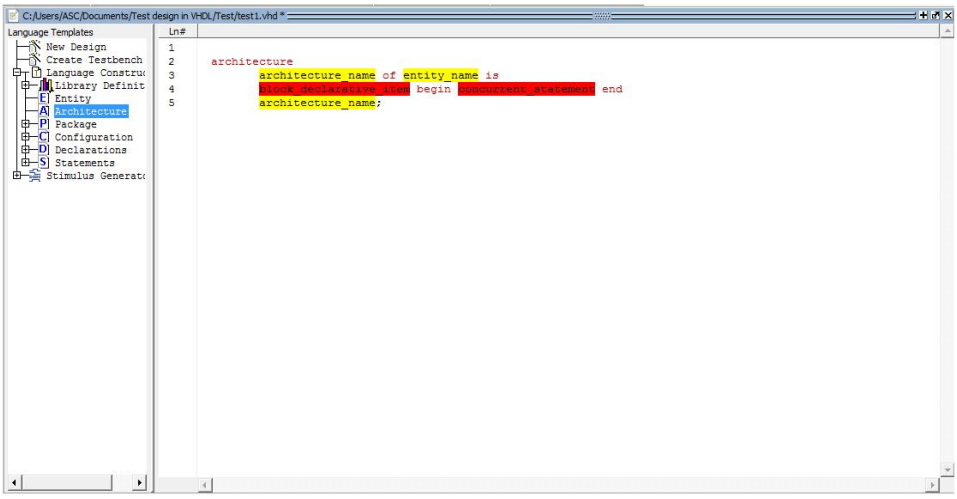




I detta nya fönster (vänster) kan man skapa nya filer eller lägga till filer i projektet. När man väljer att skapa en ny fil (höger) kan man välja filtyp under ’Add file as type’ så filnamnet får en korrekt ändelse. Efter ’OK’, öppnas en ny flik med namnet ’Projekt’ i Workspace fönstret i det övre vänstra hörnet innehållande den nya filen. Dubbelklicka på den nyskapade filen och ett editorfönster öppnas till höger (se bild nedan).

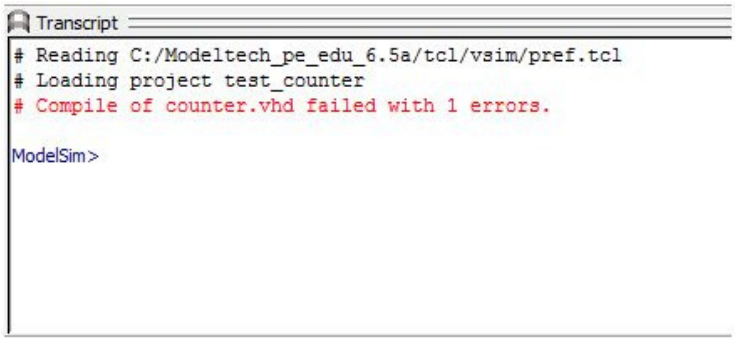


Nästa steg är att skriva VHDL kod. Notera att editorn färgsätter VHDL kommandon, datatyper och övriga reserverade ord för enklare avläsning. Genom att högerklicka på editor fönstret och välja ’Show Language Templates’ kan man lägga till enkla mallar/kodkonstruktioner till designen genom att dubbelklicka på den valda mallen.



10.2 KOMPILERING AV KOD

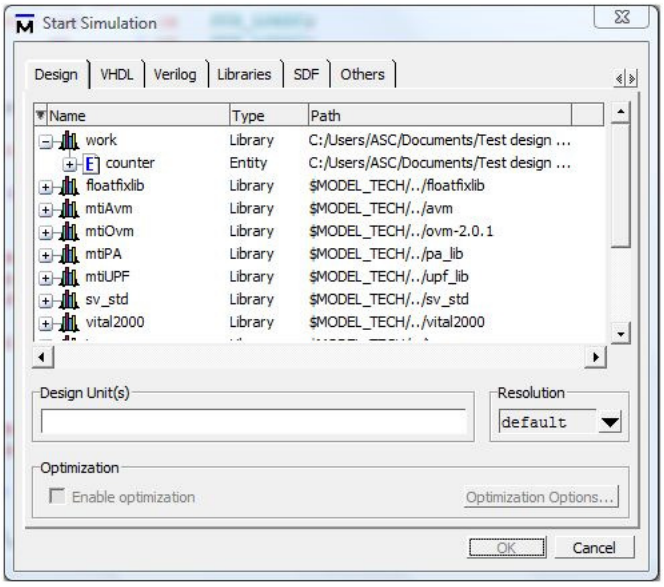
Innan simulering av komponenten kan ske måste denna kompileras. Detta kan göras via menyn ’Compile’ eller genom att högerklicka på vhdl filen i projektmappen (har man fler filer som skall kompileras under ett projekt kan detta göras genom att välja ’Compile All’). Om kompileringen går helt felfritt visas inga felmeddelanden (beroende på vilken version man kör, kan det även visas ett meddelande i form av: `Compile of fil_namn was succesful`). Om däremot ett/flera fel förekommer vid kompileringen visas detta med röd text i ’Transcript’ fönstret (se bild nedan).



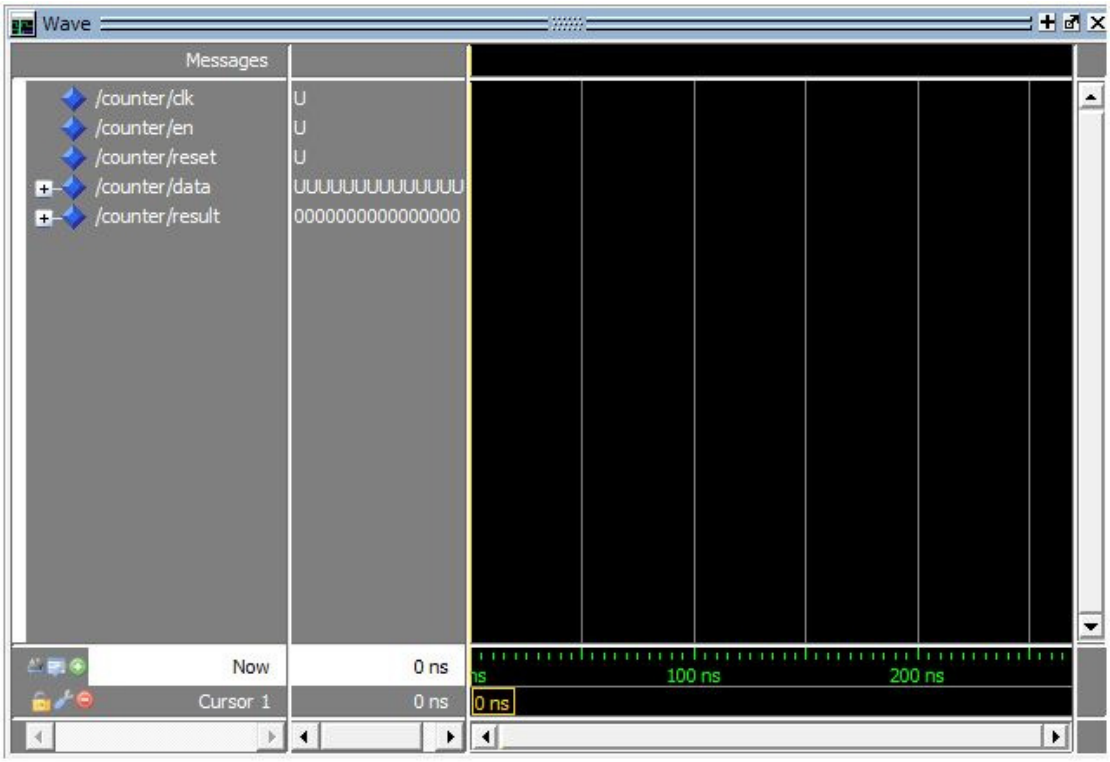
Dubbelklickar man på den röda texten, öppnas ett nytt fönster som ger mer detaljerad information (skrivet i rött) om vilken rad samt de syntax fel som skall rättas till. Dubbelklickar man på förklaringen så markeras motsvarande rad och textavsnitt i koden (text editorn) som skall rättas till.

10.3 SIMULERING

Efter felfri kompilering är det dags for simulering av komponenten. Välj **Simulate => Start Simulation**. Ett nytt fönster öppnas (nedan) och man väljer den entitet som skall simuleras. Dessa hittas under *work* mappen (tryck på ’+’ tecknet till vänster för att visa alla entiteter samt dess arkitektur).



När man valt sin entitet för simulering visas ett annat fönster med namnet ’Objects’. Här visas alla de portar och signaler som tillhör design hierarkin. För att få en användning av dessa skall de flyttas över till Wave editorn, där de kan studeras grafiskt. Markera de signaler och portar av intresse, högerklicka och välj **’Add => To Wave => Selected Signals**’, varav ett nytt wave fönster öppnas med de valda signalerna. Genom att högerklicka på signalerna kan man ändra bla. visningsläge från binär till decimal form, färgsätta mm.



Innan simulering startas är samtliga portar och signaler odefinierade. Simuleringskommandon skrivs i ’Transcript’ fönstret där signaler tilldelas ett värde genom: `force signal_namn värde`. Ex. `force signal_a 1010`.

Man kan även ange en tid för signaltilldelningen:

```
force signal_a 0 0, 1 100ns, 0 200ns
```

Kommandot ovan talar om att signal_a sätts till värdet 0 när tiden är 0, den får värdet 1 vid tiden 100ns och värdet 0 igen vid tiden 200ns.

Repeat kommandot kan vara användbar vid skapande av en klocksignal, se exempel:

```
force signal_a 0 0, 1 50ns -repeat 100ns
```

Kommandot ovan skapar en symmetrisk klocksignal med perioden 100ns.

Force kommandot kör inte själva simuleringen, utan sätter endast värden till signaler och portar. För att köra igång simuleringen skriver man enligt följande:

```
run  [<simuleringstid>]
```

10.4 .DO filer

Att skapa in stimuli till sin design enligt metoden ovan är mer lämpligt för små kretsar, men vid större system kan detta bli en tröttsam process. Istället för återupprepande ’force’ kommandon, skapas en testbänk som genererar lämpliga insignaler till komponenten. Hela simuleringsprocessen automatiseras genom skapandet av en .do fil (kan skapas i windows anteckningar) som bygger på samma enkla mall som exemplet nedan visar:

```
# Close existing ModelSim simulation
quit -sim

# Create libraries
 if {[file exist work] ==0} {exec vlib work}


# Compile top level files
vcom -93 -explicit -work work "../Source/Vhdl/alu.vhd"
vcom -93 -explicit -work work "../Source/TB/alu_tb.vhd"

# load simulation
vsim -t 1ps work.alu_tb

# Set waveform display
do alu_wave.do

# Run simulation
run 600 us
```

Vi förklarar .do filen ovan rad för rad. Samtliga rader som börjar med ett ’#’ tecken ses som en kommentar och ignoreras vis exekvering. Det allra första kommandot i filen stänger alla tidigare simuleringar, om sådana finns, innan den aktuella simuleringen påbörjas. Nästa del i koden undersöker om en arbetskatalog (work) finns till hands där de kompilerade filerna läggs. Om sådan saknas, så skapas ett.
Nu kommer vi till själva filkompileringen. Observera att ordningen på kompileringen spelar stor roll då denna exekveras sekventiellt. I kort så står det att filen alu.vhd skall kompileras följt av dess motsvarande testbänk alu_tb.vhd. Dessa filer kompileras via vhdl standard -93, och finns i work-mappen med beskriven sökväg. Nästa steg är att påbörja simuleringen. Här anges den minsta diskreta tid på tidsaxeln (i vårt fall 1 picosekund) samt den fil som anropas i work mappen. För att få fram den resulterande grafen så öppnas wave fönstret via do kommandot. Till sist anger vi simuleringstiden.

11 VHDL KOMPONENTER

Tri State Buffer

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

entity tristate is
    Port( DATA_IN : in  STD_LOGIC_VECTOR(3 downto 0);
          EN      : in  STD_LOGIC;
          DATA_UT : out STD_LOGIC_VECTOR(3 downto 0));

end tristate;

architecture behavior of tristate is
begin

  process(DATA_IN, EN)
  begin
       if EN='1' then
         DATA_UT <= DATA_IN;
       else
         DATA_UT <= (others =>'Z');
       end if;
  end process;

end behavior;
```

2:4 Avkodare

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

entity avkodare is
    Port( I : in   STD_LOGIC_VECTOR(1 downto 0);
          O : out  STD_LOGIC_VECTOR(3 downto 0));
end avkodare;

architecture behv of avkodare is
begin
    process (I)
    begin
       case I is
              when "00" => O <= "0001";
              when "01" => O <= "0010";
              when "10" => O <= "0100";
              when "11" => O <= "1000";
              when others => O <= "XXXX";
       end case;
    end process;

end behv;
```

Heladderare

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

entity heladder is
    Port ( a    : in    STD_LOGIC;
           b    : in    STD_LOGIC;
           cin  : in    STD_LOGIC;
           cut  : out   STD_LOGIC;
           s    : out   STD_LOGIC);
end heladder;

architecture Structure of heladder is

begin
      s   <= a XOR b XOR cin;
      cut <= (a AND b) OR (a AND cin) OR
     (b AND cin);

end Structure;
```

n bitars adderare

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

entity n_bit_add is

Generic(n: integer :=8);
  Port(  A, B: in  STD_LOGIC_VECTOR(n-1 downto 0);
         Cin : in  STD_LOGIC;
         S   : out STD_LOGIC_VECTOR(n-1 downto 0);
         CUt : out STD_LOGIC );

end n_bit_add;

architecture behavioural of n_bit_add is

signal total: STD_LOGIC_VECTOR (n downto 0);
signal carry: STD_LOGIC_VECTOR (n downto 0);

begin
  carry <= (0=>Cin, others=>'0');
  total <= ('0'& A) + ('0'& B) + carry;
  S     <= total(n-1 downto 0);
  Cut   <= total(n);
end behavioural;
```

D latch

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

entity D_latch is
    Port( DATA_IN :  in  STD_LOGIC;
          EN      :  in  STD_LOGIC;
          DATA_UT :  out STD_LOGIC);
end D_latch;

architecture behv of D_latch is
begin
    process(DATA_IN, EN)
    begin
       if (EN='1') then
             DATA_UT <= DATA_IN;
              end if;
    end process;
end behv;
```

D flip-flop

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

entity d_ff is
    Port( DATA_IN :  in  STD_LOGIC;
          CLK     :  in  STD_LOGIC;
          DATA_UT :  out STD_LOGIC );
end d_ff;

architecture behav of d_ff is
begin
    process(DATA_IN, CLK)
    begin
       if (CLK='1' and CLK'event) then
           DATA_UT <= DATA_IN;
       end if;
    end process;
end behav;
```

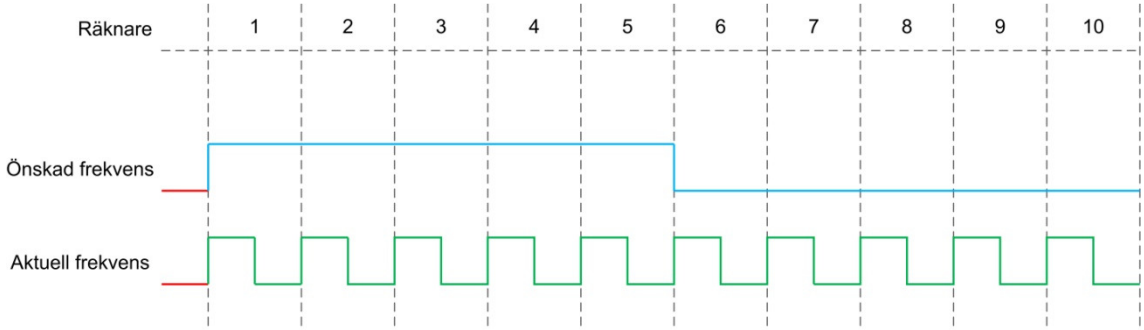
Räknare med parallell laddning

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

entity counter is
   Port (LD      : in STD_LOGIC;
         RST     : in STD_LOGIC;
         CLK     : in STD_LOGIC;
         DATA_IN : in STD_LOGIC_VECTOR(15 downto 0);
         UT      : out STD_LOGIC_VECTOR(15 downto 0));
end counter;

architecture beh_count of counter is

signal result: STD_LOGIC_VECTOR(15 downto 0) :=
(others => '0');
 begin
  process(CLK,RST)
    begin
      if RST='1' then
         result <= (others =>'0');
      elsif (CLK'event and CLK ='1') then
          if LD='1' then
             result <= DATA_IN;
          else
             result <= result + '1';
          end if;
      end if;
  end process;

UT <= result;
end beh_count;
```

12 ÖVRIGT

12.1 MANIPULATION AV KLOCKFREKVENS

I detta avsnitt kommer vi visa hur man använder sig av en räknare för att minska en klocka till önskad frekvens. Vi kan tänka oss att vi har en klockingång på 10Hz som vi vill få ner till 1Hz, enl. bild nedan:



Detta görs med två enkla steg:

- För att bestämma den nya frekvensen $\frac{Aktuell\ frekvens}{Önskad\ frekvens}$ = *gränsvärde för räknaren.* Om räknaren når detta gränsvärde, nollställs denna.

- Att bestämma ’duty cycle’ (= den tidsintervall inom vilket systemet/klockan är i aktivt läge). Vi sätter duty cycle till 50%, vilket motsvarar en ”fyrkantspuls”. $\frac{Det\ beräknade\ gränsvärdet}{2}$

Vi stoppar in våra värden och får: $\frac{10Hz}{1Hz} = 10$ = X”A” (gränsvärde för räknaren)

50% duty cycle blir då: $\frac{10}{2} = 5$.

För att vidare förtydliga detta visas ett vhdl exempel på nästa sida.

```
signal  CLK_2   : STD_LOGIC;
signal  counter : STD_LOGIC_VECTOR(23 downto 0);

architecture rtl of slow_clock is
Begin
-------------------------------------------------------
-- Slowing the clock down to 2Hz
-------------------------------------------------------
    process(RST,CLK_24MHz)
    begin
        if (RST='1') then
            CLK_2 <= '0';
            counter <= ( others => '0' );
        elsif rising_edge (CLK_24MHz) then
            counter <= counter + '1' ;
               if counter < X"5B8D80" then
                   CLK_2 <= '1' ;
               else
                   CLK_2 <= '0' ;
               end if;
               if counter = X"B71B00" then
                   counter <= ( others => '0' );
               end if;
        end if;
    end process;
```

Duty cycle:

$$\frac{B71B00}{2} = 5B8D80$$

$$\frac{Aktuell\ frekvens}{Önskad\ frekvens}$$

Innan vi skriver vår architecture så måste två signaler skapas. Den ena döper vi till CLK_2 vilket kommer motsvara vår långsammare klocksignal och den andra signalen är för räknaren (Observera att denna är en 24 bitars vektor). Så, varför sätter vi vår räknare till 24 bitar. Jo, vi använder oss av en 24MHz klocka och önskar en frekvens på 2Hz. Om vi stoppar in dessa värden i formeln på föregående sida får vi ett HEX värde av ”B71B00”. Vi vet sedan digitalteknikens grunder att varje HEX värde motsvarar 4 bitar, därmed 4x6 = 24 bitar. Nu när detta är avklarat tittar vi närmare på processen i koden. Då detta är en klockberoende process, skriver vi en asynkron reset (RST), detta i fall vårt system/klocka låser sig så kan en omstart göras. Vid nästa kodsektion talas det om att räknaren ökar med ’1’ på varje positiv klockflank. Därefter skall duty cycle sättas. Vi har valt att skapa en fyrkantspuls (se formel på föregående sida) dvs. B71B00 / 2 = 5B8D80 och då räknaren har ett mindre värde sätts vår nya klocka (CLK_2) till 1, om annat till 0. Om räknaren når upp till gränsvärdet ”B71B00” nollställs denna. Det är viktigt att nollställa räknaren, annars kommer denna räkna upp till ”FFFFFF”.

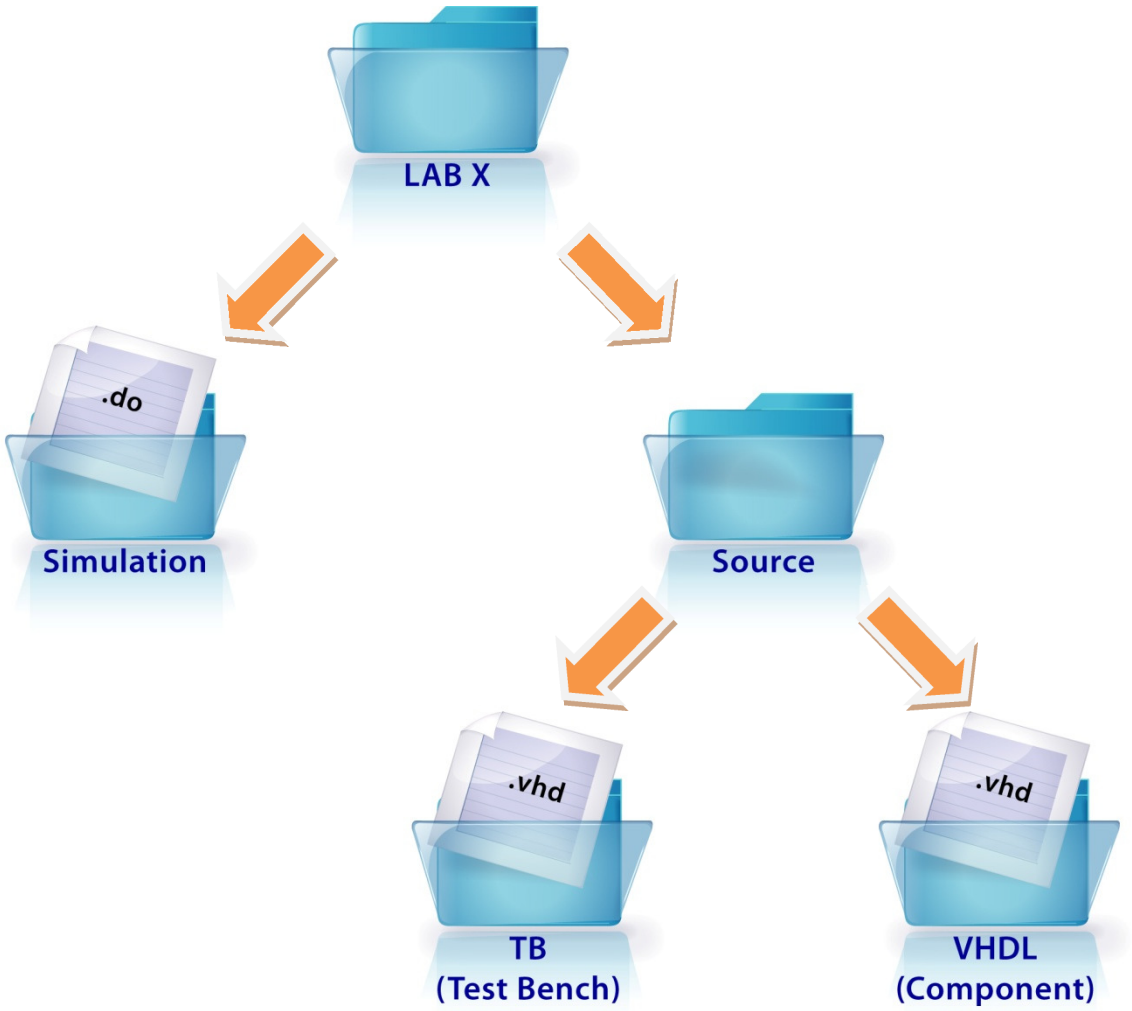
Om duty cycle skall vara 50% kan vi göra en enklare kod genom att invertera signalen CLK_2 (CLK_2<= not CLK_2;) varje gång räknaren når halva periodtiden dvs 5B8D80 i vårt fall. På så sätt behövs enbart ett HEX-värde att hantera om frekvensen skall ändras.

12.2 RESERVERADE NYCKELORD I VHDL

Det finns ett stort antal reserverade identifierare i VHDL som **ej** kan användas som egendefinierade namn. Dessa listas nedan:

abs	**else**	**map**	**range**	**then**
access	**elsif**	**mod**	**record**	**to**
after	**end**	**nand**	**register**	**transport**
alias	**entity**	**new**	**reject**	**type**
all	**exit**	**next**	**rem**	**unaffected**
and	**file**	**nor**	**report**	**units**
architecture	**for**	**not**	**return**	**until**
array	**function**	**null**	**rol**	**use**
assert	**generate**	**of**	**ror**	**variable**
attribute	**generic**	**on**	**select**	**wait**
begin	**group**	**open**	**severity**	**when**
block	**guarded**	**or**	**signal**	**while**
body	**if**	**others**	**shared**	**with**
buffer	**impure**	**out**	**sla**	**xnor**
bus	**in**	**package**	**sll**	**xor**
case	**inertial**	**port**	**sra**	
component	**inout**	**postponed**	**srl**	
configuration	**is**	**procedure**	**subtype**	
constant	**label**	**process**		
disconnect	**library**	**protected**		
downto	**linkage**	**pure**		
	literal			
	loop			

12.3 GENERELL STRUKTUR FÖR MAPP HIERARKIN



Denna struktur skall följas vid laborationerna.