Förberedande datorlektioner i Digitalteknik

Petter Källström och Oscar Gustafsson

7 februari 2022

Inledning

När man ska göra digitalteknik av en CPLD, så är den rätta vägen att

1. *konstruera* logiken (rita blockdiagram)
2. *skriva* VHDL-kod för blocken
3. *simulera* konstruktionen (koden) i ModelSim för att hitta eventuella fel och *rätta till* dem (genom att göra om tidigare steg)
4. *syntetisera* koden när den väl fungerar, för att få en binärfil som innehåller konfigurationen för CPLDn
5. *bränna* (programmera) CPLDn
6. *koppla upp* kretsen och verifiera funktionen.

Dessa steg beskrivs i kapitel 1

I denna lektionsmanual finns tre enkla uppgifter, D1 till D3, där du ska öva på olika delar av stegen, VHDL och verktygen. Exakt vilka moment som ingår beror dock på vilken kurs du läser. Dessutom påverkas innehållet av om VHDL-delen av kursen ges på distansform eller på plats.

TSEA22 och TSEA52 : Alla uppgifter är obligatoriska i sin fulla form. Två lektionstillfällen är schemalagda. Förslaget är att försöka hinna med uppgift D1 och D2 på första lektionen, och uppgift D3 på andra.

TSEA51 : Uppgift D1 är obligatorisk. Ett tillfälle är planerat. Uppgift D2 och D3 görs i mån av tid.

Distansform : Delen att bränna och koppla upp ingår inte. Istället läggs fokus på att kunna använda mjukvaran på distans. Läs noga igenom appendix A.

Obligatoriska uppgifter ingår som delmoment i laboration 3, men ska lösas på lektionstid.

Alla uppgifter har *skal*, som bl.a. innehåller en s.k. testbänk för simulering. Lektionerna är upplagda som en laboration, men med detaljerade instruktioner för hur ni löser simulering och syntes.

Uppgift D1 är en enkel enpulsare, med tydliga steg-för-steg-instruktioner genom hela kedjan. I uppgift D2 och D3 ska ni konstruera två varianter på räknare. Ev. resterande tid används med fördel för laborationsförberedelse. Observera dock att inga laborationsuppgifter kan redovisas under lektionstid.

Lektionen utgår ifrån att ni sitter i Windowsmiljö i laborationssalen. Senare vid laborationsförberedelse är fjärrinloggning på universitetets datorer ett alternativ till att besöka salarna. I föreläsningsvideo 9.8 Fjärrinloggning på skolans Windowsdatorer visas hur universitetets Windowsdatorer med tillhörande program kan användas hemifrån. I appendix A finns en kortfattad beskrivning hur universitetets Linuxdatorer kan användas på distans.

Innehåll

1 De olika stegen **2**

2 Uppgift D1: Enpulsare **2**

3 Simulering **2**
- 3.1 Labskalet 3
- 3.2 Starta ModelSim 3
- 3.3 Designfilen 3
- 3.4 Simulera 4
- 3.5 Åh nej, något är fel! 5

4 Syntes **5**
- 4.1 Starta verktyget ISE 6
- 4.2 Starta ett nytt projekt 6
- 4.3 Syntetisera 7

5 Bränning och uppkoppling **7**

6 Uppgift D2: Räknare **8**
- 6.1 ModelSim-trix 8
- 6.2 Redovisning 9

7 Uppgift D3: Dekadräknare **9**

A Arbeta på distans **11**
- A.1 Inloggning via ssh 11
- A.2 Inloggning via ThinLinc . . . 11
- A.3 Arbeta i Linux 12

B Kort VHDL-introduktion **13**
- B.1 Signaler, processer och kombinatoriska grindar 13
- B.2 Indentering 14
- B.3 Exempel: ROM 14
- B.4 Exempel: Räknare 16
- B.5 Exempel: Tillståndsmaskin . 17

1 De olika stegen

Att *konstruera* logiken (blockdiagram, ekvationer etc) förväntas ni kunna från övriga lektioner och laborationer.

Att *skriva VHDL-kod* ingår i laborationernas genomförande. Ni kommer dock att få skriva lite VHDL på lektionerna.

För att *simulera* används verktyget ModelSim. Där testar ni att er skrivna VHDL beter sig rätt.

För att *syntetisera* används verktyget Xilinx ISE. Det tolkar om VHDL-filen till AND-OR-logik och register samt genererar den data som behövs för att konfigurera hårdvaran. De CPLD:er vi har i labbet har konstruerats av företaget Xilinx.

Notera att programmen ModelSim och Xilinx ISE inte har med varandra att göra. Båda utgår från samma VHDL-fil, men gör helt olika saker med den.

Först när man *bränner* och *kopplar upp*, så kommer man i kontakt med hårdvaran. Det hör också till laborationerna.

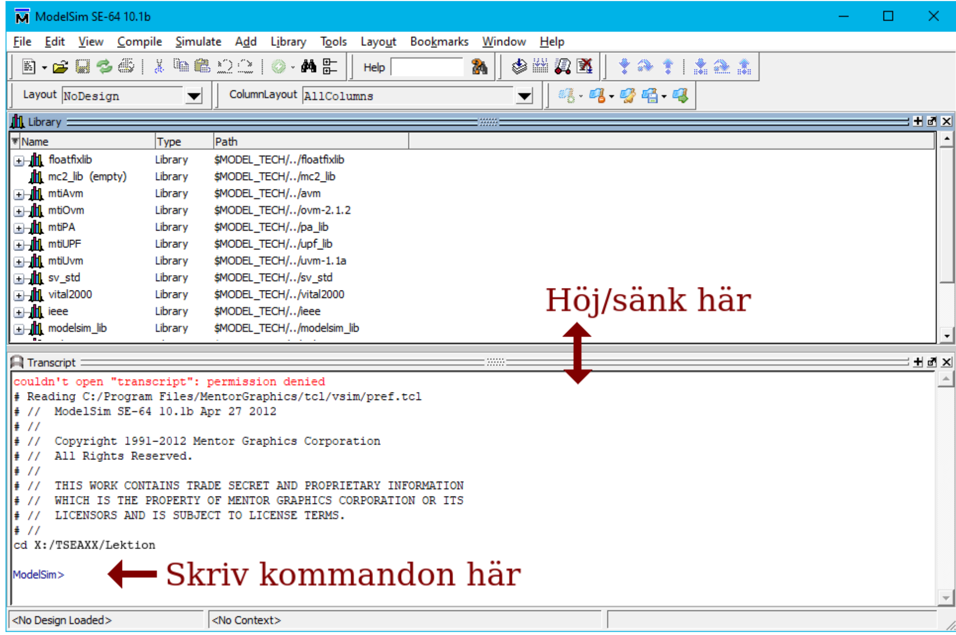
2 Uppgift D1: Enpulsare

Uppgift D1 beskrivs i stegen om simulering, syntes och uppkoppling nedan.

3 Simulering

Simulering innebär att konstruktionen sätts i en slags virtuell testmiljö. Man stoppar in lämpliga ettor och nollor på ingångarna, och kollar att det som kommer ut beter sig rätt. Detta kan göras manuellt, men det är tidsödande. Enklare är att göra det via en fördefinierad *testbänk*. Testbänken ansluter till den konstruktion som testas (eng. "design under test", DUT). Även testbänken är skriven i VHDL, men använder sig av funktioner i språket som inte går att ha i en CPLD.

Det här med att simulera VHDL-kod *är* ett erkänt krångligt kapitel, och lite av en överkurs i det här skedet. Därför har vi skapat dessa laborationsskal, med bland annat färdig testbänk.



Figur 1: ModelSims fönster, efter att ha navigerat till mappen X:\TSEAXX\Lektion.

3.1 Labskalet

Börja med att ladda ner laborationsskalet för lektionen, det är en .zip-fil med tre mappar. I mappen `enpulsare` finns fem filer. `first_time.do` och `rerun.do` är hjälpskript, som vi återkommer till. `enpulsare.vhdl` är själva design-filen, som är påbörjad, men saknar själva logiken. `enpulsare_tb.vhdl` är testbänken. Slutligen har vi `syntetisera.sh` som kan användas för syntes i linux-miljö.

Packa upp zip-filen och placera filerna i lämplig katalog, t.ex. `X:\TSEAxx\Lektion`.[1]

3.2 Starta ModelSim

Starta ModelSim från startmenyn. Öppna menyn och skriv “ModelSim”, så får ni upp förslag.

Stäng eventuell välkommen-ruta.

Navigera till mappen där filerna för enpulsaren finns: File › Change Directory. Figur 1 visar ungefärligt resultat, och visar var ni så småningom ska skriva in kommandon för att simulera.

3.3 Designfilen

Öppna `enpulsare.vhdl` i valfri redigerare för textfiler, t.ex. Programmers Notepad, eller Notepad++. ModelSim och Xilinx ISE har varsin inbyggd redigerare (ModelSims rekommenderas dock ej).

Den korrekta konstruktionen visas i figur 2. VHDL-filen har tom arkitektur, precis som i skalen till laboration 3 och 4. Uppgiften är att fylla den med kod som definierar insidan av kretsen.

Komplettera VHDL-filen med den **in**korrekta koden nedan (inklusive den avsiktliga buggen). *Skriv av* koden – man lär sig bättre av det, och copy-paste riskerar att få med sig osynliga skräptecken som kan ge svårlösta problem vid kompilering.

[1] I Windows kan .zip-filen se ut som en komprimerad katalog, där det går att redigera filer, men simulering och syntes kräver att filerna verkligen är uppackade.

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

entity enpulsare is
  port(clk, reset : in std_logic;
       x : in std_logic;
       u : out std_logic);
end entity;

architecture behav of enpulsare is
   signal x_sync : std_logic;
   signal x_sync_old : std_logic;
begin
-- The two D-flip flops
process(clk, reset) begin
   if reset = '1' then
      x_sync <= '0';
      x_sync_old <= '0';
   elsif rising_edge(clk) then
      x_sync <= x;
      x_sync_old <= x_sync;
   end if;
end process;

u <= x_sync and x_sync_old;

end architecture;
```

Paketet `std_logic_1164` definierar typen `std_logic`, som är våra binära bitar.

Porten definierar pinnarna på kretsen.

Arkitekturen beskriver insidan av kretsen. Ovanför `begin` deklarerar vi två interna "sladdar".

Kommentarer börjar med `--`.

Med processen definierar vi två D-vippor. Båda klockas med `clk` och nollställs av `reset`.
Den ena läser från `x` och skriver till `x_sync`.
Den andra läser från `x_sync` och skriver till `x_sync_old`.

Slutligen skapar vi AND-grinden som skriver till utsignalen `u`. **Notera att det finns en avsiktlig bugg här**, som testbänken ska få hitta.

3.4 Simulera

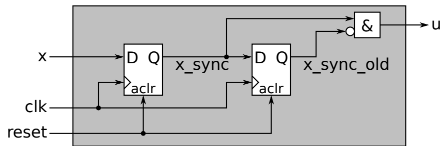
ModelSim beter sig helt olika beroende på om det har en design "laddad" i simulatorkärnan eller inte. Nu är det dags att kompilera koden, testbänken, samt ladda in bägge i simulatorn.

Gå till ramen "transcript" i ModelSim, och kör kommandot:

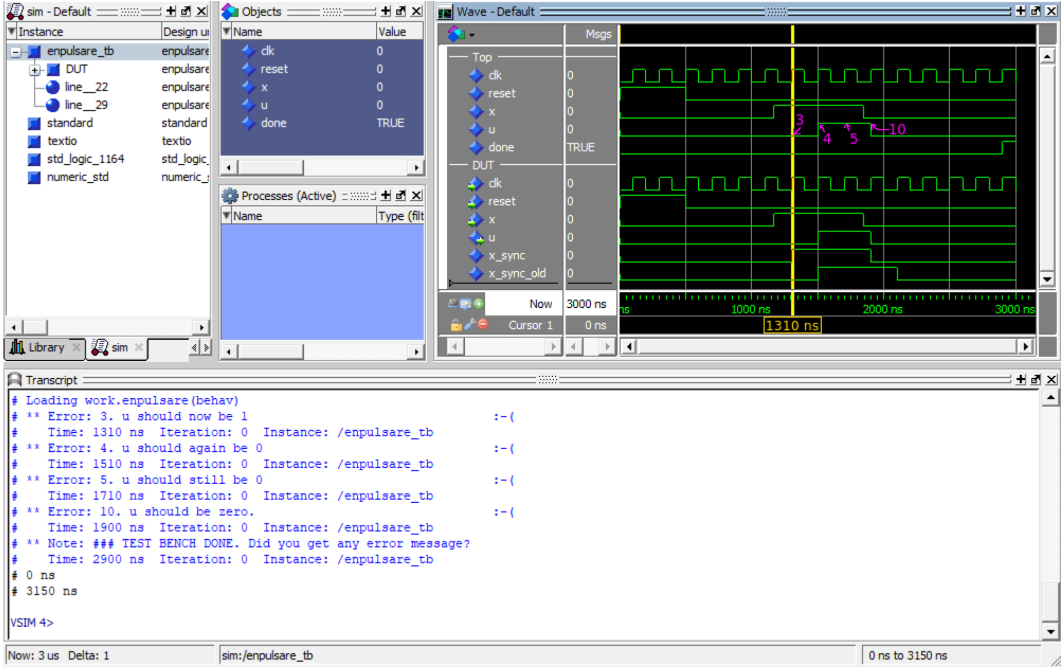
`do first_time.do`

Detta kompilerar VHDL-filerna, laddar in testbänken i simulatorn (och via den även er design), lägger till filerna i ett vågfönster, och kör simuleringen. När ModelSim laddar in designen så kommer fönstren att hoppa runt lite.

Om ni får kompileringsfel, så försök rätta till det, och kör sedan kommandot igen (det går att använda [↑] + [↵] för att köra förra kommandot igen). Om ni bara får en röd utskrift som säger att något är kompileringsfel, så kan det hjälpa att dubbelklicka på den för att se mer i detalj vad kompilatorn klagade på.



Figur 2: Vår enpulsare, med interna strukturen.



Figur 3: Simuleringsresultatet, innan buggen rättats. De rosa siffrorna är tillagda i efterhand för att visa var felutskrifterna sker.

Resultatet ses i figur 3.

3.5 Åh nej, något är fel!

Testbänken har rapporterat fyra felmeddelanden (test nummer 3, 4, 5, 10). Det är i regel bra att ta hand om första felmeddelandet först.

Dubbelklicka på första felutskriften, så får ni upp en gul markör i vågfönstret.

Enpulsaren ska ju detektera att `x` just har tryckts ner (alltså inte att `x` *är* nertryckt), genom att se till att `x_sync` är 1, samtidigt som den var 0 förra klockcykeln. Så har vi gjort i designen (lilla cirkeln i figur 2). Men det stämmer ju inte vid gula markören när `x_sync` = 1 och `x_sync_old` = 0. Istället kommer utsignalen när `x_sync` = 1 och `x_sync_old` = 1.

Våra register verkar stämma, men utsignalen `u` gör det inte. Det verkar vara något i ekvationen som genererar `u`. Lägg in den glömda inverteraren på raden för `u` i designfilen:

```
u <= x_sync and not x_sync_old;
```

Spara filen.

I ModelSim måste vi nu simulera om. Det kan vi göra med samma kommando som förut, men när designen nu är laddad så finns ett snabbare alternativ:

```
do rerun.do
```

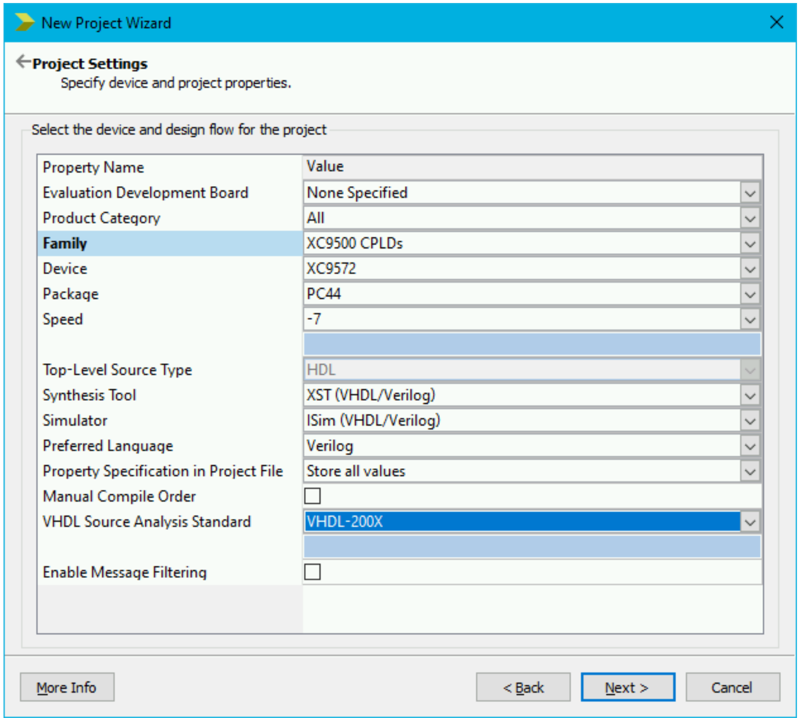
Detta kompilerar om koden och startar om simuleringen "på plats" i simulatorkärnan.

Om ni får kompileringsfel, så rätta dessa och kör om (↑ + ↵).

Nu ska alla felutskrifter vara borta. Dags att börja syntetisera.

4 Syntes

När kod *syntetiseras* så översätts den till en binärfil, som kan brännas in på en CPLD. Syntesprogrammet tolkar er kod, och gör efter bästa förmåga om den till and/or-uttryck och register. Dessutom bestäms hur dessa ska spridas ut på CPLDns resurser, samt hur de ska kopplas ihop. Slutligen skapas den *.jed*-fil som definierar innehållet i CPLDn.



Figur 4: ISE Project Wizard där bland annat CPLDn specificeras.

4.1 Starta verktyget ISE

Öppna Xilinx ISE, som ligger i startmenyn, under ”ISE Design Suit”, och heter Project Navigator – välj 64-bitars-versionen.

Om rutan “Did you know...” kommer upp, så stäng den.

4.2 Starta ett nytt projekt

Knappen New Project... . Sätt ett bra namn, t.ex. “enpulsare”. *Location* och *Working Directory* sätter ni till där ni har era filer. Se till att Top-level source type är HDL. Klart – klicka på Next > .

Skriv namnet först

OBS! När ni skriver in ett namn så kan Location ändras. Skriv därför namnet först.

Nu ska vi berätta för syntesprogrammet vilken CPLD vi har enligt figur 4. Notera speciellt följande inställningar:

- Family = XC9500 CPLDs
- Device = XC9572
- Package = PC44
- Speed = –7
- VHDL Source Analysis Standard = VHDL-200X

Klart – Next > , sen Finish .

Nu *kan* en ny VHDL-fil skapas, men det är ju redan gjort. Istället ska den befintliga läggas till. Välj Project » Add Source... . Därmed har er konstruktion blivit importerad i projektet, och ni ser den som *toppmodul*.

4.3 Syntetisera

Klicka på den gröna "play"-ikonen ▶, eller välj Process » Implement Top Module. Det tar ett tag. Om allt går bra, så ska ni få upp ett fönster med syntesrapport. Om inte, så kan ni klicka på summa-ikonen Σ eller välj Project » Design Summary/Reports.

Design Summary dyker upp som en flik i ISE, med summa-ikonen. Till vänster i den hittar ni Design Overview » CPLD Fitter Report.

I **CPLD Fitter Report** så finns några noterbara delar (i menyn till vänster).

Under **Summary**, så får man se en summering av hur många makroceller, produkttermer etc. som används. I ett and-or-uttryck, så är det en produkt-term per AND-grind, och en makrocell består av flera produkttermer och en OR-grind. Vår enpulsare använder två D-vippor (kallat register), samt en produktterm.

Under **Equations** så ses en lista på alla ekvationer som används i CPLD:n. Denna lista är i regel svår att förstå, då den använder konstiga uttryck, som ".LFBK". Med kvalificerade gissningar går det ofta att räkna ut vad som sägs.

Under **Pin List** får ni reda på vilka in- och utgångar som hamnar var på den fysiska kapseln. Skriv ner dessa (`x`, `u`, `clk` och `reset`), då ni kommer att behöva dem senare.

Kolla runt lite på olika delar här. Speciellt så vill ni titta på ev. fel och varningar. Dessa finns antingen via flikarna längst ned eller under Summary där det finns en länk i tabellen.

Varningar

Vissa varningar går inte att få bort och en del är okej att ha. Det ni ska hålla speciellt utkik efter är varningar om **latches** och **sensitivity list**. Dessa kan innebära att er konstruktion inte fungerar likadant i simuleringen som när ni har syntetiserat. Det **får inte** finnas några sådana varningar för er konstruktion!

Om ni får varningar av denna typ så behöver ni ändra i koden samt simulera om för att bekräfta att simuleringen fortfarande är korrekt innan ni syntetiserar. Det **måste** med andra ord vara samma version av koden som ger korrekt simuleringsresultat och syntetiseras utan relevanta varningar.

Det kan ibland vara till hjälp att syntetisera koden för att hitta buggar i simuleringen som orsakats av sämre skriven kod. Den typen av varningar som nämns ovan kan vara orsaken till att kretsen inte fungerar som tänkt och genom att syntetisera så pekas tveksamma rader ut (som ni ändå behöver skriva om förr eller senare).

Slutligen så har syntesverktyget producerat en .jed-fil (som heter t.ex. `enpulsare.jed`) i projektmappen. Denna fil innehåller information om hur brännarprogrammet ska konfigurera CPLDn, så att den fungerar enligt er beskrivning.

5 Bränning och uppkoppling

Nu är det dags att bränna och koppla upp kretsen. Leta upp den .jed-fil som har skapats. Kopiera den.

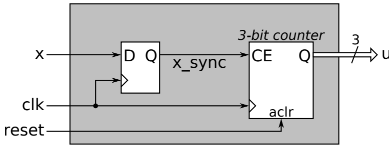
Gå till L:\, skapa en tom map och döp den till ditt liu-ID. Klistra sedan in .jed-filen i den mappen.

Om L: saknas, så får ni montera den som nätverksenhet. Vid brännarstationen står det hur man gör.

Gå till brännardatorn, ungefär i mitten av rummet. Be lektionsassistenten hjälpa er bränna in .jed-filen i en krets.

Koppla därefter in kretsen på laborationsplattan som är färdigkopplad (så när som på era pinnar). När det fungerar, be att få redovisa lektionsuppgiften för assistenten. Uppgifterna ska redovisas på lektionerna, men kan i undantagsfall även redovisas vid laborationstillfällen.

Om det är kö till brännardatorn eller uppkopplingen, så kan ni påbörja lektionsuppgift D2 under tiden.



Figur 5: Räknaren i uppgift D2, med dess interna struktur.

6 Uppgift D2: Räknare

Nu ska ni implementera en räknarkrets, illustrerad i figur 5. Räknaren har tre bitar. En insignal, `x`, ska synkroniseras (utan reset), och därefter användas som *count enable* (CE). Räknarvärdet ska skickas ut på utsignalen `u`.

Utgå från skalet `counter` (i lektionens .zip-fil). Filen `counter.vhdl` innehåller viss hjälp. Det finns även en räknare bland exemplen längst bak i denna lektionsmanual. Slutför filen (lägg gärna in några avsiktliga fel i den, och se vad som händer).

I ModelSim måste ni först avsluta förra simuleringen, om den är igång (leta reda på funktionen under Simulate-menyn). Navigera till nya mappen, simulera, rätta till, simulera om etc, tills allt fungerar.

Syntetisera kretsen. Hur mycket resurser används?

6.1 ModelSim-trix

Testbänkarna i den här laborationsserien kommer att hjälpa er hitta och åtgärda fel som kan vara tidsödande att lösa genom uppkoppling. Dock kan testbänkarnas utskrift vara lite kryptisk att förstå, och ännu svårare att komma fram till var buggen finns.

Det är som sagt bra att börja med första felrapporten, och att ett dubbelklick på felutskriften placerar en gul markör i vågformen.

Man kan **zooma** x-axeln, om man t.ex. tydligare vill se vad som händer inför ett rapporterat fel. Till det används scrollhjulet som en knapp i vågformen. Prova att "dra en linje" med scrollhjulet

- **upp åt vänster** ger *zoom full* (allt visas)
- **upp åt höger** ger *zoom out* – längden på linjen spelar roll
- **ned åt vänster eller höger** ger en *zoom box*.

Man kan också zooma genom att Ctrl+scrolla, funktionen kan uppfattas icke-intuitiv.

Man kan **färga signaler**, genom att högerklicka på dem, välja "Properties", sen "Colors" och slutligen välj en bra färg. Det är bra om man har väldigt många signaler i sin vågform, och vill kunna fokusera på bara ett fåtal. I figur 6 har signalen `x` färgats gul.

Med långa namn så kan det bli svårt att läsa vad signalerna heter. Välj då den lilla ikonen "Toggle Leaf Names" direkt till vänster under vågformen.

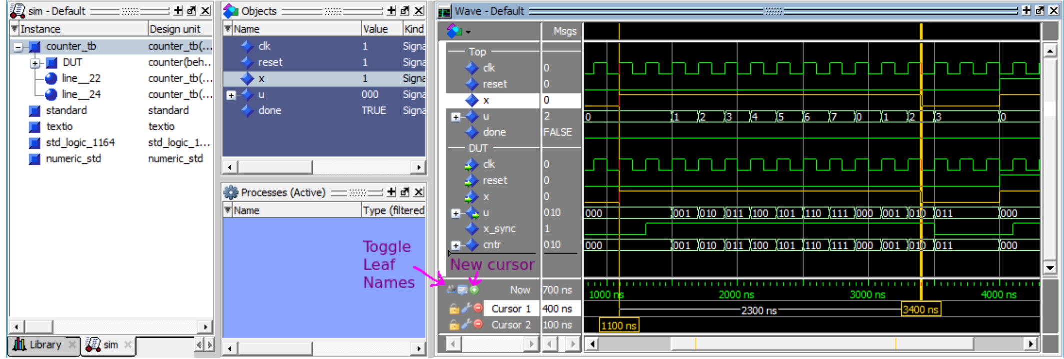
En signal som representerar positiva heltal (som ju fallet är för räknaren), kan formateras decimalt istället för binärt: Högerklicka på signalen, Radix > Unsigned.

Ni kan **spara vågformatet** som en .do-fil, vilket är behändigt när man gjort massor av inställningar. Välj File > Save. Senare kan ni lägga till alla signaler med format genom att köra kommandot (om filen nu heter wave.do):

```
do wave.do
```

Notera att detta lägger till signalerna på nytt i vågfönstret. Ni kan manuellt skriva in `delete wave *` överst i .do-filen, för att först ta bort alla befintliga signaler.

ModelSim har en bra **tab completion** när ni skriver kommandon i transcript-fönstret. Den föreslår resten av kommandot eller filnamnet. Ni kan alltså skriva `do f`tab istället för hela `do first_time.do`, vilket sparar tid.



Figur 6: Simuleringsresultat av räknaren i uppgift D2, med många av trixen.

Man kan lägga till ytterligare en gul markör ("cursor"), genom att klicka på det lilla gröna pluset till vänster under vågfönstret. Med två markör är det lättare att mäta tid. Man flyttar en markör genom att dra i den. Genom att hålla muspekaren på en övergång mellan 0 och 1, så kan markören "snappa" till den exakta tidpunkten.

6.2 Redovisning

Uppgift D2 består av att diskutera kod, syntes- och simuleringsresultat.[2] Visa också att du kan panorera och zooma effektivt i vågfönstret. Uppgiften kan redovisas samtidigt som uppgift D3. För att redovisa flera uppgifter samtidigt, så behöver flera fönster av ModelSim och Xilinx ISE vara öppna.

7 Uppgift D3: Dekadräknare

I den tredje uppgiften, illustrerad i Figur 7, ska ni implementera en dekadräknare som kan räkna både uppåt och nedåt. Den måste naturligtvis ha 4 bitar, eftersom den ska räkna till 9. Speciellt gäller att

- den ska räkna när count enable (`CE`) = '1'
- riktningen är upp när `up/`$\overline{\texttt{down}}$ = '1', annars ned
- den ska nollställas synkront av `clear` = '1'
- den synkroniserar alla insignaler
- den har en asynkron reset (aktivt hög), som dock inte gäller synkroniseringen av insignaler.

Funktionen Modulo-10 ger specifikt att

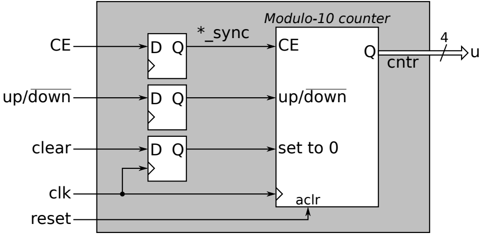
- Uppräkning från 9 ger 0 ($9 + 1 \bmod 10 = 0$)
- Nedräkning från 0 ger 9 ($0 - 1 \bmod 10 = 9$).

Fundera på vad som ska hända om `CE` och `clear` båda är aktiva samtidigt. Hur ska det formuleras i VHDL?

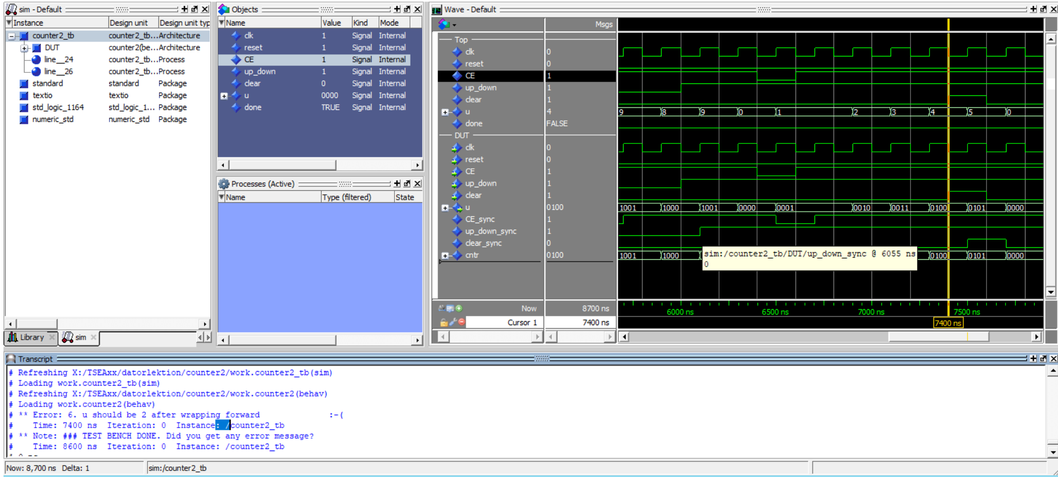
Skapa denna räknare. Lägg gärna in någon bugg för att se hur testbänken reagerar. Simulera allt i ModelSim, rätta till buggar o.s.v.

Figur 8 visar Wave-fönstret där testbänken hittat ett fel. Så här kan felet lokaliseras i detta fall: Testbänken förväntar sig att räknaren ska ha kommit till värde 2 vid tidpunkt

[2] Under laborationen kommer naturligtvis även uppkopplingen att ingå



Figur 7: Dekadräknare i uppgift D3.



Figur 8: Simulering av uppgift D3. Testbänken har hittat ett fel.

7400 ns. Genom att klockcykel för klockcykel jämföra med kommentarerna i testbänken, så kan man se att `u` börjar fela straxt efter att `up_down` sätts till 1. `u` räknar upp från 8 till 9 direkt, trots att `up_down_sync` fortfarande är 0 (`u` borde alltså räkna *ner*). Aha – räknaren verkar använda sig av `up_down` istället för `up_down_sync`.

Slutligen, syntetisera dekadräknaren i Xilinx ISE. Hur mycket resurser går åt?

A Arbeta på distans

Om ni arbetar på distans så kan ni logga in på någon Linux-dator för att editera VHDL-filer, simulera eller syntetisera.

För enkelhets skull går detta appendix endast igenom när ni ansluter till linux-datorer. Om ni bara vill flytta filer mellan er egna dator och konton på universitetet, så ta en titt på länken nedan. Er hemkatalog heter då t.ex. `/home/liuid123/`.

Det finns två olika sätt att koppla upp sig: antingen direkt från egen dator via ssh till en linuxdator i Muxen eller via ThinLinc till en universitetsdator och sedan via ssh till datorerna i Muxen. ThinLinc är typiskt snabbare men tar över hela skärmen, medan att koppla sig via ssh direkt gör att programmen dyker upp som vilket annat program som helst.

På följande sida finns mer om fjärrinloggning:
`https://www.student.liu.se/studentstod/itsupport/linuxdatorsalar/fjarrinloggning?l=sv`

Snabbinfo: `sftp://ssh.edu.liu.se`.

A.1 Inloggning via ssh

För att koppla upp direkt med ssh hemifrån så behövs det först anslutas till LiU interna nät via VPN. All trafik kommer då gå via LiUs datorer, så det kan vara bra att koppla ner sig när arbetsdagen är slut. Läs vidare på:
`https://www.student.liu.se/itsupport/vpn-forticlient?l=sv`.

Datornamn

Ni ska använda datorerna i Muxen 1 och 2. Dessa heter `muxenY-0XX.ad.liu.se`, där `Y` = 1 eller 2 och `0XX` = 001 till 016. Ett exempel är `muxen2-013.ad.liu.se`.

Windows

Om er dator har Windows, så kan ni installera t.ex. MobaXterm, `https://mobaxterm.mobatek.net/`, som har stöd för SSH med grafik. Det finns även andra verktyg. I MobaXterm så väljer ni menyn Sessions > New session. I fönstret väljer ni SSH och en av datorerna enligt boxen Datornamn ovan. Port 22 (som bör vara förinställt). Under "Advanced SSH Settings", så ser ni till att X11-forwarding och Compression är markerat.

Linux

Om datorn har Linux, så öppnar ni en terminal och skriver:
`ssh -XC liuid123@muxenY-0XX.ad.liu.se`
(byt ut `user123` mot användarnamn och `muxenY-0XX` mot ett av datornamnen beskrivna i boxen ovan).

Mac

Om datorn är en Mac, så kan programmet XQuartz, `https://www.xquartz.org/`, användas och därifrån köra samma ssh-kommando som beskrivs i Linux. Har datorn OSX 10.5 eller äldre så finns motsvarande XQuartz redan installerat.

A.2 Inloggning via ThinLinc

Om ni väljer att använda ThinLinc, `https://www.cendio.com/`, så behöver ni ansluta till `thinlinc.edu.liu.se`. Öppna där en terminal (högerklicka på skrivbordet och välj Open Terminal Here) och följa instruktionerna för Linux ovan.

Håll koll på ThinLincs "Popup menu key"

Innan ni trycker på Connect i ThinLinc, så tryck på Options... och kom ihåg vilken tangent som står som "Popup menu key". Denna tangent kan ni trycka på för att få upp en ThinLinc-meny där ni t.ex. kan välja bort helskärmsläget eller minimera ThinLinc och därmed komma tillbaka till det vanliga skrivbordet.

A.3 Arbeta i Linux

Väl inloggad i Muxen via terminal kommer vissa saker bli annorlunda, jämfört med Windows.

Hemkatalogen ligger nu i `/home/liuid123/`.

A.3.1 Textredigerare

I Linux finns t.ex. `emacs`, `gedit` eller `vim`. Ni kan även använda `atom`, men då måste ni lägga till modulen genom kommandot `module add prog/atom`.

A.3.2 Starta ModelSim

För att starta ModelSim måste ni först köra kommandot `module add prog/modelsim`. Därefter startas programmet genom `vsim &` (&-tecknet på slutet gör att ni kan fortsätta använda terminalen).

Ibland kan det vara tidsödande att köra grafiska program via ssh och/eller ThinLinc. Ett alternativ då är att istället köra simulering och syntes i textläge i terminalen.

Kommandot `vsim -c` startar ModelSim i terminalläge. `vsim -c -do first_time.do` kör do-filen `first_time.do`. Notera att det går att skriva kommandon på samma sätt som i ModelSim, så det går att t.ex. starta om simuleringen på samma sätt som i ModelSim. Det går alltså att första skriva `vsim -c` i terminalen och sedan `do first_time.do` när ModelSim-prompten dyker upp. Eftersom `first_time.do` lägger till vågformer och dessa inte kan visas i textläge så kommer ni får några varningar om detta, men dessa kan ignoreras. Att vågformerna inte kan visas är så klart ett problem om det finns fel, men för att snabbt testa *om* det fungerar (och för att visa att det fungerar) så är det användbart.

A.3.3 Starta Xilinx ISE

För att starta Xilinx ISE måste ni först köra kommandot `module add prog/xilinx_ise`.[3] Därefter startas programmet genom `ise &`.

Equations finns bara på Windows

Equations fungerar bara på Windows. Under Linux hittar ni all information under CPLD Fitter Report (text), men är inte lika enkelt läsbar som på Windows.

Även syntesen kan köras i enbart textläge. Då är det dock ett antal steg som behövs. För att underlätta har vi ett script, `syntetisera.sh`, som kör dessa syntessteg och skriver ut alla rader med fel eller varningar. Ibland skriver dock verktygen ut fler rader än skriptet visar, så ni kan behöva läsa loggfilerna. Filnamnet för de olika loggfilerna skrivs ut så att ni ska veta var ni ska läsa.

Scriptet körs med `./syntetisera.sh`. Det krävs dock att filen markerats som körbar. Ni kan behöva köra kommandot `chmod u+x syntetisera.sh` som sätter exekverbarflaggan (`+x`) för er som användare (`u` = user).

[3] På vissa system kan modulen xilinx_ise förstöra licensinställningar för ModelSim, då kan ni öppna en ny terminal till ModelSim.

B Kort VHDL-introduktion

Inför första labben: Läs igenom kapitel 1.4 i läroboken samt läs igenom följande exempel. Dessa exempel utgör en sammanfattning av den VHDL-kod som ni nu bör klara av att förstå, samt skriva själva.

B.1 Signaler, processer och kombinatoriska grindar

Här är ett alternativt synsätt på VHDL och kopplingen till hårdvara.

När vi deklarerar signaler, så är det sladdar vi deklarerar (med en eller flera bitar). Även in- och ut-signaler i entiteten är sladdar.

Varje sladd får bara kopplas till en TTL-utgång, annars blir det kortslutning. På samma sätt får varje signal bara skrivas till på ett ställe i koden (en hel process räknas som ett ställe).

Vid syntes, så kommer verktyget att göra sitt bästa för att förverkliga det beteende du ber om, med de resurser som finns i CPLD'n (typiskt grindlogik och vippor).

Utsignalen `u` i enpulsaren tilldelas `x_sync and not x_sync_old` *hela tiden*. Syntesverktyget kommer därför skapa[4] kombinatorisk grindlogik som motsvarar det aktuella uttrycket, och koppla utgången till sladden. På så sätt uppdateras sladden *hela tiden*, så syntesverktyget lyckas perfekt.

Den interna signalen `x_sync_old` får värdet av `x_sync` endast vid `rising_edge(clk)`, alltså stigande klockflank. För att syntesverktyget ska lyckas återskapa detta, så använder den en D-vippa, vilken exakt uppfyller behovet.

Som framgår i exemplen nedan, så kan man även inkludera en del logik i processer. Syntesverktyget kommer då att använda sig av logiska grindar i kombination med vippor för att åstadkomma motsvarande beteende.

För att inte äventyra syntesverktygets möjlighet att förverkliga koden, så *ska* processer se ut så här:

```
process(clk, reset) begin
  if reset = '1' then
    -- async reset signals here
  elsif rising_edge(clk) then
    -- assign signals here. You can use e.g. if statements.
  end if;
end process;
```

Om man inte har asynkron reset, så går man direkt på `if rising_edge(clk) then`.

Så här definieras processens beteende (förenklat):

- Varje gång någon signal i sensitivity-listan (`process(sensitivity)`) ändras, så exekveras processen.
- Processen exekveras uppifrån och ner, och kan innehålla if-satser m.m.
- Om en signaler blir tilldelad, så uppdateras den med det (senast) tilldelade värdet när processen är klar. Man kan alltså inte tilldela ett värde och sedan läsa resultatet på raden nedanför.

Det här gör att processer är ideala för att konstruera "massa logik följt av D-vippor". Med bakgrund av detta, vilket beteende beskriver en process, om ni glömt `rising_edge(clk)`?

[4] "skapa" syftar på att använda befintlig resurs i CPLD'n

B.2 Indentering

Indentering är mellanslag/tab man lägger framför varje rad, för att göra koden mer läsbar. En del textredigerare (t.ex. Emacs) har fullt inbyggt stöd för indentering. De flesta andra har hjälpligt stöd på någon nivå. Oavsett vad, så *måste* koden indenteras. Ta som vana att indentera *direkt*, medan du skriver. Laborationsassistenten kommer troligen varken godkänna eller hjälpa dig alls, om du inte har indenterad kod.

Ett problem är om man blandar olika textredigerare, med olika inställningar för tab och mellanslag. De flesta redigerare går dock att ställa in inställningar på. Försök ställa in att använda tab, med tabavstånd på t.ex. två eller fyra.

Det finns ingen exakt standard för hur det ska se ut, men utifrån grundregeln att underlätta läsbarhet har vi: Allt som "börjas", ska "slutas" på samma nivå, och det som omsluts ska indenteras ett steg till. Det är viktigt att man lätt kan *se* var t.ex. en if-sats har sina elsif, else och var den slutar. En pågående sats över flera rader, t.ex. port(...), eller when-else, bör tvärt om ha alla övriga rader en nivå in, för att visualisera att det är en fortsättning på förra raden. Figur 9 visar detta, där indenteringsnivåerna har ritats in, med förklarande namn. I ROM-exemplet nedan har jättemånga mellanslag använts för att vänsterjustera ROM-innehållet.

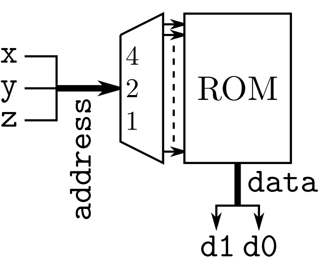
```
entity foo is
  port( ...
        ...
      );
end entity;

architecture behav of foo is
  signal ...;
  signal ...;
begin
  process(...) begin
    if ... then
      ...
    elsif ... then
      ...
      if ... then
        ...
      else
        ...
      end if;
    end if;
  end process;

  x <= ... when ... else
    ...;
end architecture;
```


Figur 9: Exempel på de osynliga linjer som finns vid indentering.

B.3 Exempel: ROM

Här följer ett exempel som visar hur ett ROM kan implementeras samt hur vektorer kan skapas och delas upp.

Följande ROM med storleken 8×2 bitar ska implementeras för det angivna minnesinnehållet.



x	y	z	d_1	d_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Motsvarande VHDL-kod är:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;  -- Package for treating numbers

entity ROM1 is
  port (x, y, z : in std_logic;
        d0, d1 : out std_logic);
end ROM1;

architecture behav1 of ROM1 is
  -- Definition of the ROM memory:
  type ROM_mem is array(0 to 7) of std_logic_vector(1 downto 0);
  constant ROM_content : ROM_mem := (0 => "00",
                                     1 => "01",
                                     2 => "10",
                                     3 => "10",
                                     4 => "00",
                                     5 => "01",
                                     6 => "10",
                                     7 => "11");

  -- Other signals:
  signal address : std_logic_vector(2 downto 0);
  signal data : std_logic_vector(1 downto 0);

begin
  -- Assign bits to the address vector.
  address(2) <= x;
  address(1) <= y;
  address(0) <= z;

  -- Read the ROM.
  data <= ROM_content(to_integer(unsigned(address)));

  -- Assign result bits
  d0 <= data(0);
  d1 <= data(1);
end architecture;
```

Förklaring till raden där ROM:et läses: `address` är en vektor av bitar, utan numerisk betydelse (exempel: `"011"` som std_logic_vector har inget med talet 3 att göra). Först gör vi om vektorn till typen `unsigned`, som är en typ som tolkar ihop bitarna som värden (exempel: `"011"` som unsigned motsvarar nu talet 3). Sedan används `to_integer` till att göra om formatet till typ integer, vilket är ett heltal utan koppling till bitar (exempel: 3). Slutligen är `ROM_content(`n`)` den n:te raden i ROM:et, där n är den integer vi just konverterat fram.

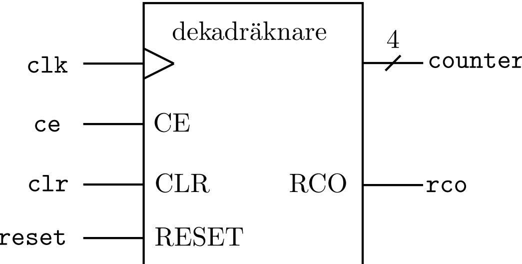
Alternativt används en `with-select`-sats enligt:

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

entity ROM1 is
  port (x, y, z : in std_logic;
        d0, d1 : out std_logic);
end ROM1;

architecture behav2 of ROM1 is
  -- Signals
  signal address : std_logic_vector(2 downto 0);
  signal data : std_logic_vector(1 downto 0);

begin
  -- Assign bits in address
  address <= x & y & z;  -- The "&" means concatenate, or merge

  -- Implement a ROM directly:
  with address select data <=
    "00" when "000" | "100",
    "01" when "001" | "101",
    "10" when "010" | "011" | "110",
    "11" when others;  -- "others" = all not covered cases.

  -- Assign result bits.
  d0 <= data(0);
  d1 <= data(1);
end architecture;
```

Här har raderna slagits ihop till en rad på kodord, för att visa hur det kan göras. Generellt försämras dock läsbarheten av detta format, då innehållet inte kommer i naturlig ordning.

B.4 Exempel: Räknare

Nästa exempel visar hur en dekadräknare kan implementeras i VHDL. Den har asynkron reset, `reset`, synkron clear, `clr`, count enable, `ce` och ripple carry out, `rco`.



Nedan följer koden för räknaren. Vad händer om clr och ce är aktiva samtidigt i koden nedan?

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;        -- Package for arithmetic

entity counter is
   port(clk, reset : in std_logic;
        ce : in std_logic;
        clr : in std_logic;
        q : out unsigned(3 downto 0);
        rco : out std_logic);
end entity;

architecture behav of counter is
  signal q_int : unsigned(3 downto 0); -- Unsigned: Bit-vector as number
begin

  -- the counter core:
  process(clk, reset) begin
    if (reset = '1') then                 -- asynchronous reset
      q_int <= to_unsigned(0, 4);
    elsif rising_edge(clk) then
      if (clr = '1') then                 -- synchronouse clear
        q_int <= to_unsigned(0, 4);
      elsif (ce = '1') then               -- Count enable
        if q_int = 9 then
          q_int <= "0000";
        else
          q_int <= q_int + 1;             -- arithmetic operation
        end if;
      end if;
    end if;
  end process;

  -- output signals:
  rco <= '1' when ((ce = '1') and (q_int = 9)) else '0';
  q <= q_int;
end architecture;
```

För att räkna behövs paketet `ieee.numeric_std.all` som inkluderas på rad tre. På första raden i architecture definieras `q_int` som är räknarvärdet. `q_int` måste vara `unsigned`, eftersom det representerar tal 0 till 9, och det krävs fyra bitar.

Den asynkrona reset-signalen ska bara användas för att manuellt återstarta räknaren och kopplas typiskt till en resetknapp. Om räknaren ska nollställas under drift används den synkrona clear-signalen.

B.5 Exempel: Tillståndsmaskin

Det avslutande exemplet visar hur en enkel tillståndsmaskin kan beskrivas. Vidare läsning refereras till kapitel 9.3 i Hemert.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;        -- Package for arithmetic

entity FSM is
  port(clk, reset : in std_logic;
       x : in std_logic;
       q : out std_logic_vector(1 downto 0));
end entity;

architecture behav of FSM is
  signal x_sync : std_logic;

  type my_states_t is (ADAM, BERTIL, CEASAR);
  signal my_state : my_states_t;
begin
  -- sync input:
  process(clk, reset) begin
    if reset = '1' then
      x_sync <= '0';
    elsif rising_edge(clk) then
      x_sync <= x;
    end if;
  end process;

  -- FSM process:
  process(clk, reset) begin
    if reset = '1' then
      my_state <= ADAM;
    elsif rising_edge(clk) then
      if x_sync = '1' then
        if my_state = ADAM then
          my_state <= BERTIL;
        elsif my_state = BERTIL then
          my_state <= CEASAR;
        else -- CEASAR
          my_state <= ADAM;
        end if;
      else
        if my_state = ADAM then
          my_state <= CEASAR;
        elsif my_state = BERTIL then
          my_state <= ADAM;
        else -- CEASAR
          my_state <= BERTIL;
        end if;
      end if;
    end if;
  end process;

  -- Output:
  q <= "00" when my_state = ADAM else
       "01" when my_state = BERTIL else
       "11"; -- CEASAR
end architecture;
```