Digital Konstruktion TSEA43

Ingemar Ragnemalm 2001,
Olle Seger 2003-, olles@isy.liu.se

19 december 2012

Innehåll

Kapitel 1

Introduktion till VHDL

1.1 Inledning

VHDL är ett hårdvarubeskrivande programmeringsspråk. Förkortningen VHDL står för VHSIC Hardware Description Language, där VHSIC i sin tur står för Very High Speed Integrated Circuit. [1] Ett hårdvarubeskrivande språk används för att programmera programmerbara kretsar på hög nivå, utan att behöva bry sig så mycket om hur konstruktionen ser ut på grindnivå, och i synnerhet utan att behöva handgripligen koppla in varenda grind. Olika språk abstraherar olika mycket. Med VHDL kan man abstrahera väldigt mycket, men man har också möjligheten att vara relativt maskinnära.

Denna lilla skrift avser att ge dig de mest fundamentala grunderna i VDHL, nog för att göra enklare konstruktioner, men är inte en komplett lärobok till språket. Den är avsedd för kursen Digital Konstruktion. Vi förutsätter att du läst digitalteknik, men att du inte stött på VHDL tidigare. Om du har tidigare erfarenhet av VHDL är troligtvis denna introduktion överflödig så du kan gå direkt till labbuppgifterna i kapitel 4.

Innan vi går in på språket så skall vi ta en titt på vad programmerbar logik är för något.

1.2 Programmerbara kretsar

Tidigare byggde man digitala apparater av kretsar med fast funktion, grindar, vippor, räknare, multiplexrar med mera, sådana komponenter som du hittar i TTL-familjen och som du använde i labbarna i Digitalteknik. För stora serier gjorde man förstås specialkonstruerade kretsar, men prototyper och mindre serier byggdes mycket på grindnivå. I DK-kursen kommer ni att ha tillgång till TTL-familjen och andra standardkomponenter med fast funktion (minnen, AD/DA-omvandlare med mera), men dessutom programmerbara kretsar. En programmerbar krets kan

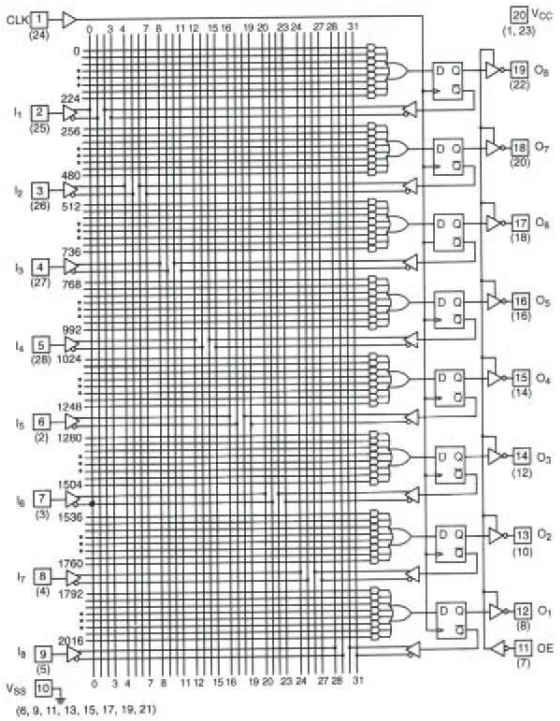
[1] Very high speed i VHSIC är egentligen irrelevant, för språket kräver naturligtvis inte att kretsarna är extremt snabba.

programmeras till att utföra mer eller mindre komplicerade uppgifter, och därmed ersätta flera TTL-kretsar. Den enklaste formen är PAL, Programmable Array Logic. En PAL-krets är en mycket enkel programmerbar krets, som i sin enklaste form bara kan implementera ganska enkla kombinatoriska nät, eller, om vippor ingår, enkla sekvensnät.

En något mer sofistikerad form är PLD, Programmable Logic Device. Både PAL och PLD innehåller en programmerbar array, samt ett grindnät från arrayen till en rad utgångar. Dessa utgångar kan vara utgångarna från en rad vippor (som i PALen 16R8) eller så kallade makroceller, där vippan ingår i ett litet nät som tillåter viss konfigurering. Större kretstyper, CPLD (Complex PLD) och FPGA (Field Programmable Gate Array) innehåller flera block som var och ett kan motsvara en PLD. Som första exempel tittar vi på PALen 16R8. Det är en PAL med D-vippor, vilket innebär att den kan användas för att konstruera sekvensnät.

1.2.1 Exempel på PAL, 16R8

I arrayen kan lodräta ledare anslutas till de vågräta ledare som går in i grindnäten till höger. Varje lodrät signal har redan en vågrät ansluten, nämligen någon av följande: en insignal, en utsignal från en vippa, eller inverterade versioner av dessa. De vågräta ledarna som går till grindnäten är betydligt fler än vad som syns på bilden, flera ledare representeras med en linje, men lodräta är blott 32.

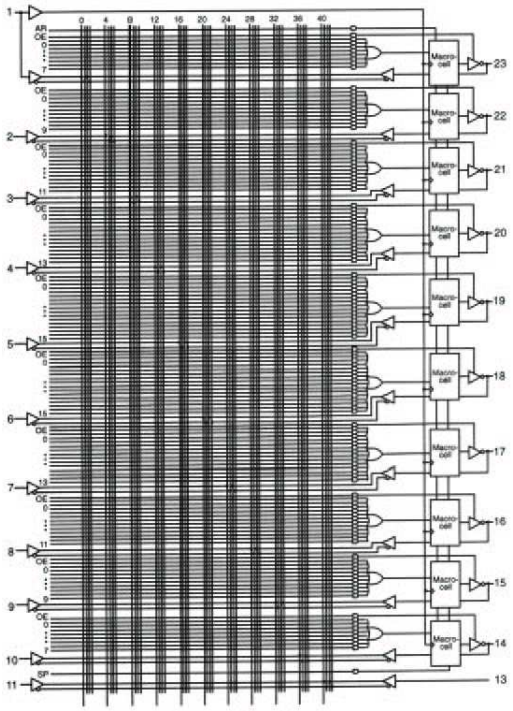


Figur 1.1: PAL 16R8.

När man ansluter lodräta ledare med vågräta ansluter man signaler till ingångar på grindarna. De grindar som inte ansluts får värdet 1 genom interna pull-up-motstånd. Vi måste allså inte ansluta alla ingångar i detta fall. Varje grindnät är en rad AND-grindar vars utsignaler matar en OR-grind, vars utsignal i sin tur går till en D-vippa.

1.2.2 22V10

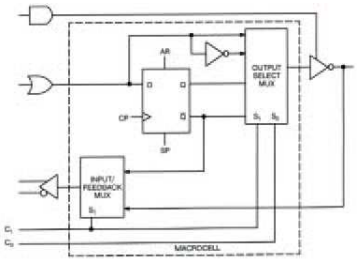
Låt oss nu gå vidare till PLDn 22V10. 22 är antalet ingångar och 10 antalet utgångar. De 10 utgångarna överlappar ingångarna, så 22 och 10 är det maximala antalet ingångar respektive utgångar, men summan av dem kan inte överstiga 22. Man inser lätt att man aldrig har 22 ingångar, för vad uträttar en krets som saknar utgångar?



Figur 1.2: PLD 22V10.

Bilden är grötigare än 16R8, men enda skillnaden mot 16R8 är att det är fler in- och ut-signaler, 44 lodräta ledare i stället för 32, och D-vipporna har ersatts av makroceller. Signalerna från D-vipporna kan nu ersättas av in/utgångens värde beroende på hur makrocellen konfigureras. En makrocell i 22V10 är uppbyggd så här:

Som du ser i figuren innehåller makrocellen en vippa och två multiplexrar. Konfigurationssignalerna, C_0 och C_1 i figuren, styr multiplexrarna och anger därmed en av följande funktioner: C_0 anger om makrocellen skall bete sig som en D-vippa.



Figur 1.3: Makrocellen i 22V10.

Värdet 0 anger att D-vippa skall användas. Värdet 1 anger att signalen skall passera förbi vippan, dvs att cellen används för kombinatoriskt nät

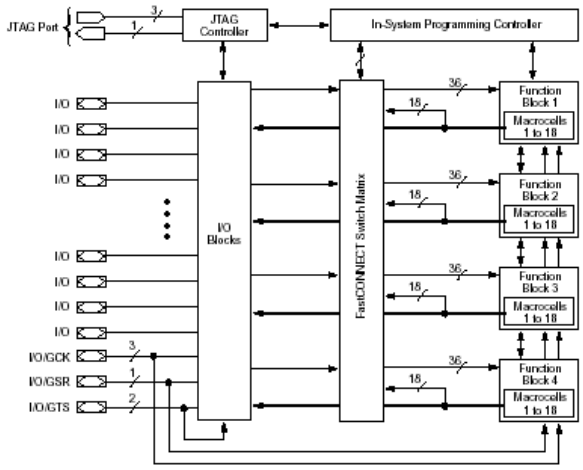
C_1 anger om signal eller inverterad signal skall användas. Man har många möjligheter att invertera signaler, både i makrocellerna och på ingångarna till grindnäten. Detta gör att de flesta någorlunda enkla kombinatoriska uttryck som man behöver kan implementeras i grindnätet vid varje makrocell. Med bara 10 vippor är 22V10 ganska begränsad. Bokstaven V i 22V10 står för att grindnäten är av variabel storlek, dvs att de tio näten är olika stora. Detta framgår av figuren. Grindnäten kallas för produkttermer, eftersom de är ett tvålagers nät av AND-grindar följt av en OR-grind. Ingångarna till dessa kallas produkter resp. termer, analogt med multiplikation och addition. I industrin bygger man ofta sina konstruktioner med bara en enda programmerbar krets. Man tar helt enkelt till en som är så stor att den räcker. 22V10 rymmer uppenbart inte ett DK-projekt, och det är inte meningen heller. Räkna med att använda flera, och ett antal specialiserade kretsar dessutom, i synnerhet minnen. En lämplig användning för en 22V10 är att implementera ett kombinatoriskt nät eller ett sekvensnät. Du ersätter ett antal grindar och vippor med en krets.

1.2.3 CPLD Xilinx 9572

Vi går vidare ytterligare ett steg och tittar på en CPLD (Complex PLD). Det visar sig att det effektivaste sättet att bygga större PLD-er är att kombinera flera stycken mindre PLD-er mha av ett förbindningsnät. Anledningen till detta är att AND-planet annars tenderar att bli underutnyttjat. Figur 1.4 visar en XC9572, 72 makroceller, uppbyggd som 4 st 36V18 med ett förbindningsnät, som gör att en del signaler i en 36V18 kan göras tillgängliga globalt. Denna krets kommer att användas i kursen från och med VT2004. I samma kretsfamilj finns också 9536 och 95108 att tillgå hos oss.

1.2.4 Minimikrav och goda råd

Nu hoppas vi förstås att du är nyfiken och sugen på att använda programmerbara kretsar för att göra eleganta konstruktionslösningar, men vi inser att det finns de



Figur 1.4: Blockschema över Xilinx XC9572.

som hellre vill använda det de redan kan och kanske tycker VHDL verkar jobbigt. Låt mej därför notera några krav och riktlinjer:

- Ni måste programmera och använda minst en 9572 som del av ert projekt!
- 9572 kan ersätta ett antal TTL-kapslar, så tänk på att den tid du förlorar på att programmera och bränna tar du igen på att slippa vira så mycket! En justering av din konstruktion kan ofta göras utan att ändra några ledare alls!
- När ett delproblem kan lösas lika bra med en standardkrets så använd för all del den, om ni föredrar det.
- Simulera era konstruktioner innan ni bränner.

1.3 En första smak på VHDL

Du är säkert van vid programmeringsspråk, som Java, Ada och C/C++. Dessa språk är designade för sekvensiell exekvering. VHDL är radikalt annorlunda. Eftersom det beskriver hur en programmerbar krets skall bete sig så beskriver den *parallella* förlopp. Även om syntaxen påminner om Ada så kom ihåg att det är en helt annan sak! Tänk på digitalteknik, vippor och grindar, inte på programsteg!

Ett typiskt VHDL-program har följande struktur (där ord med versaler är reserverade ord):

```
ENTITY namn1 IS
beskrivning av in- och utgångar
END ENTITY namn1;
```

```
ARCHITECTURE namn2 OF namn1 IS
beskrivning av interna signaler
BEGIN
beskrivning av funktion
END ARCHITECTURE namn2;
```

En `ENTITY` beskriver konstruktionens in- och utgångar, medan `ARCHITECTURE` beskriver beteendet. En komplicerad konstruktion kan byggas upp av många delblock, men vi kommer att koncentrera oss på det enkla fallet där ett `ENTITY`/`ARCHITECTURE`-par skall läggas i en 9572.

Ett komplett `ENTITY`-block kan se ut så här:

```
ENTITY in2out1 IS
PORT(in1, in2: in bit;
     out1: out bit);
END ENTITY in2out1;
```

och ett motsvarande `ARCHITECTURE`-block kan se ut så här:

```
ARCHITECTURE grind OF in1out2 IS
BEGIN
  out1 <= NOT (in1 AND in2);
END ARCHITECTURE grind;
```

Jaha, så vad var det vi konstruerade? En NAND-grind!

1.4 Kombinatoriska nät med VHDL

Ovan såg du hur man gör ett trivialt kombinatoriskt nät, en enda grind. Låt oss ta en titt på vilka funktioner vi har tillgång till när vi gör kombinatoriska nät. Det handlar i första hand om Booleska uttryck. Vi har följande:

- Booleska operationer: `AND, OR, XOR, NOR, NAND, NOT`
- Tilldelning av signal: `<=`.
- Parenteser: `()`.

Detta är de mest uppenbara byggstenarna, men det finns rikligt med andra sätt att uttrycka sig. När vi ser listan inser vi att vi kunde uttryckt NAND-grinden ovan ännu enklare:

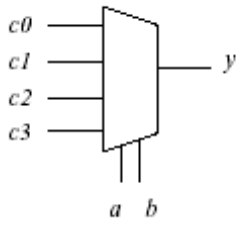
```
out1 <= in1 NAND in2.
```

Ingen av de Booleska operationerna har prioritet över någon annan. Du måste använda parenteser för att visa vilka operationer som skall göras först. Följande uttryck är inte ens tillåtet:

```
x <= a OR b AND c;
```

I stället måste parentes sättas runt den del som utförs först.

Låt oss ta ett något intressantare exempel. Du skall göra ett kombinatoriskt nät som realiserar en multiplexer som liknar den i LS153, en 1-av-4-väljare. Med signalerna `a` och `b` väljs en av `c0, c1, c2, c3` till utsignalen `y`.



Figur 1.5: Multiplexer.

Med Karnaughdiagram eller sunt förnuft tar du fram det Booleska uttrycket:

$$y = \bar{b} \cdot \bar{a} \cdot c_0 + \bar{b} \cdot a \cdot c_1 + b \cdot \bar{a} \cdot c_2 + b \cdot a \cdot c_3$$

eller i VHDL-syntax:

```
y <= (NOT b AND NOT a AND c0) OR (NOT b AND a  AND c1)
    OR (b AND NOT a AND c2) OR (b AND a AND c3);
```

I många fall kan det vara bekvämt att utnyttja uttryck som är lite elegantare än ett Booleskt uttryck. För kombinatoriska nät har vi `WITH-SELECT-WHEN` och `WHEN-ELSE`. Med dessa blir koden mer lik vad vi är vana vid från sekvensiella programmeringsspråk, men å andra sidan blir det då snabbt svårare att ha koll på resursåtgången. Vi återkommer till dem lite senare.

1.5 Sekvensnät med VHDL

När man bygger sekvensnät med VHDL så använder man begreppet `PROCESS`. I en `PROCESS` utförs inte operationer omedelbart, utan signaler uppdateras först när något villkor uppfylls - lämpligtvis uppåtflank på klocksignalen. Man specifierar alltså inte D-vippor explicit. Det kan kännas onaturligt när man kommer från digitaltekniken, men angreppsättet kommer till sin rätt när man använder VHDL mer på högnivåvis. Låt oss börja med det enklaste sekvensnät som finns. Följande kod definierar en D-vippa:

```
ENTITY de IS
  PORT(d, clk: IN bit;
       q: OUT bit);
```

```
END de;

ARCHITECTURE dvippa OF de IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
      q <= d;
    END IF;
END PROCESS;
END dvippa;
```

Den intressanta biten är så klart `PROCESS`-satsen. Argumentet `(clk)` är en så kallad *sensitivity list*, en lista av signaler som triggar uppdatering. Simulatorn använder den för att bestämma när den skall exekvera processen. Normalt har vi bara klocksignalen här. Du kan möjligen också ha med asynkron nollställningssignal.

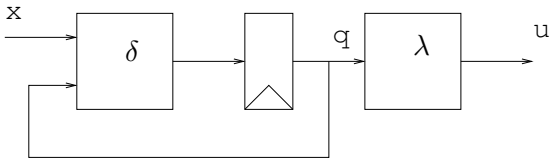
Man hör ibland att: koden inom processen utförs sekvensiellt. Detta är mycket förrädiskt, för i den slutliga PLDn finns ju ingen processor som utför programsteg. I stället analyserar kompilatorn din kod och genererar ett sekvensnät som utför funktionen.

Ordet sekvensiellt kan måhända förstås om vi *simulerar* PROCESS-satsen i t ex Modelsim. När triggvillkoret inträffar (typiskt positiv klockflank) går simulatorn igenom koden sekvensiellt och räknar ut nästa värde, utan att uppdatera, på de ingående signalerna. Uppdateringen av signalerna sker till sist i botten på PROCESS-satsen. Slutsatsen av denna lilla utvikning är att ingenstans i VHDL går det att tänka vanlig programmering.

Nu följer en `IF`-sats. Poängen med den är att få koden att trigga på en förändring av klockan, och enbart på uppåtflank.

En bit kan ha ett av två värden, `'1'` (hög, sann, +5V) eller `'0'` (låg, falsk, jord). `IF`-satsen testar på att den är hög efter förändring. Inom `IF`-satsen finns den kod som skall utföras när klockan går hög. Här skall du skriva in uttryck för att uppdatera dina D-vippor, dvs tillstånden i dina sekvensnät.

När du skall bygga ett sekvensnät kan du utgå från Digitalteknikens Moore-modell. I denna finns två kombinatoriska nät, δ och λ. λ är ett kombinatoriskt nät som genererar utsignaler enbart beroende på tillståndet, och δ genererar nästa tillstånd från tillståndet och insignalerna.

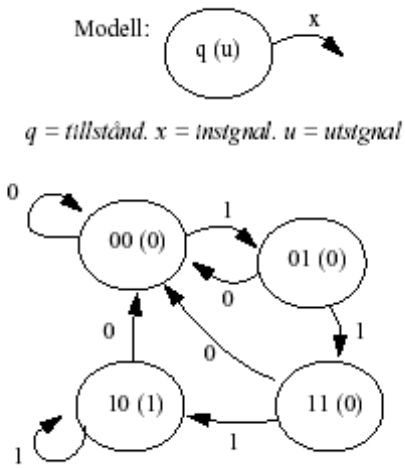


Figur 1.6: Sekvensnät enligt Moore-modellen.

Exempel: Vi vill bygga ett sekvensnät som ger utsignalen 1 när insignalen varit 1 minst tre klockcykler i rad. Annars är utsignalen 0. För detta krävs fyra tillstånd:

1. Senaste insignal var 0.
2. Senaste insignal var 1, men den innan var 0 (en etta i rad).
3. Två ettor i rad.
4. Tre ettor i rad eller mer.

Eftersom vi skall implementera detta med booleska uttryck så måste vi välja värden på de fyra tillstånden. Vi väljer 00, 01, 11, 10. Då blir sekvensnätet enligt följande figurer.



Figur 1.7: Tillståndsgraf för Moore-maskin. Enkelt sekvensnät som detekterar tre eller flera ettor på insignalen.

Nu kan vi ta fram Booleska uttryck, med Karnaughdiagram om vi vill. Uppdatering av tillstånd, dvs δ-nätet:

$$\begin{aligned} q_1^+ &= x \cdot (q_1 + q_2) \\ q_2^+ &= x \cdot \bar{q_1} \end{aligned} \tag{1.1}$$

Utsignal, dvs λ-nätet:

$$u = q_1 \cdot \bar{q_2} \tag{1.2}$$

När vi vill göra detta i VHDL så måste vi så klart deklarera tillståndsvariablerna q_1 och q_2. Det görs på följande sätt, mellan `ARCHITECTURE`-raden och `BEGIN`:

```
SIGNAL q1, q2: bit;
```

Detta behöver inte nödvändigtvis motsvara vippor. Det är först senare, när signalerna används i en `PROCESS`-sats, som detta bestäms. Nu kan vi använda `q1` och `q2` som interna signaler i konstruktionen.

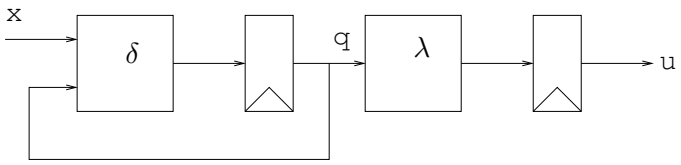
Då blir de Booleska uttycken så här:

```
q1 <= x AND (q1 OR q2)
q2 <= x AND NOT q1
u <= q1 AND NOT q2
```

Nu skall detta in i en `PROCESS`-sats. Vad tror du om detta?

```
PROCESS (clk)
BEGIN
IF rising_edge(clk) THEN
  q1 <= x AND (q1 OR q2)
  q2 <= x AND NOT q1
  u <= q1 AND NOT q2
END IF;
END PROCESS;
```

Tyvärr, detta är inte rätt. Varför? Jo, vi har lagt λ-nätet inom `IF`-satsen. Därmed säger vi att utsignalen `u` bara får uppdateras när klockan ändras. Jamen, det är väl det vi vill? Nej, tyvärr, det är det inte. Vi vill ha en utsignal som genereras av vippornas utsignaler. När dessa ändras är en annan sak. Med λ-nätet i `IF`-satsen begär vi, implicit, att få extra D-vippor på utsignalen! En signal som uppdateras vid klockpuls måste lagras i en vippa! Uttryckt schematiskt skulle vi få ett sekvensnät enligt följande figur 1.8! (Jämför med den tidigare figuren 1.6.)



Figur 1.8: Felaktigt sekvensnät pga felplacerat λ.

Men detta, däremot, fungerar:

```
PROCESS(clk)
BEGIN
IF rising_edge(clk) THEN
  q1 <= x AND (q1 OR q2)
  q2 <= x AND NOT q1
END IF;
END PROCESS;
```

```
u <= q1 AND NOT q2
```

Det finns en nackdel med detta sätt att bygga sekvensnät. Om du behöver göra någon ändring så måste du göra om det helt och hållet, med nya Booleska uttryck. Det finns andra, mer flexibla sätt att arbeta.

Fördelen med sekvensnät som det ovan är att du har mer kontroll. Tillståndet är representerat av två bitar, och uttrycken är med marginal enkla nog, så det hela kommer bara att kosta två makroceller (vippor). Abstraktion är viktigt i komplicerade konstruktioner, men är ofta en nackdel i enkla konstruktioner.

I avsnittet Mer sekvensnät nedan kommer exemplet igen i en annan form, som illustrerar dina valmöjligheter lite mer.

Här följer det kompletta VHDL-programmet för sekvensnätet ovan:

```
ENTITY trippel IS
PORT(x, clk: IN bit; u: OUT bit);
END trippel;

ARCHITECTURE one OF trippel IS
SIGNAL q1, q2: bit;
BEGIN
PROCESS (clk)
  BEGIN
  IF rising_edge(clk) THEN
    q1 <= x AND (q1 OR q2)
    q2 <= x AND NOT q1
  END IF;
END PROCESS;

u <= q1 AND NOT q2
END one;
```

Vi avslutar detta avsnitt med en rekommendation:

- Välj Moore-modellen!
- Bygg δ-nätet och därmed vipporna med en `PROCESS`-sats!
- Bygg λ-nätet som kombinatorik utanför `PROCESS`-satsen! Kom ihåg att signaler som nämns i vänsterledet inuti `PROCESS(clk)`-satsen får en vippa på sig!
- Lägg märke till skillnaden mellan ekvationerna (1.1) och (1.2). VL i (1.1) har plustecken, dvs det är signalernas nästa värde som räknas ut. Just detta åstadkoms genom att använda en `PROCESS`-sats.

1.5.1 PORT-satsen: IN, OUT, INOUT och BUFFER

I konstruktionen ovan tilldelas utsignalen `u` med den sista tilldelningen. Eftersom `u` bara tilldelas, men aldrig används, så kan den deklareras `out`. Hittills har vi bara använt `IN` och `OUT`, men det finns ytterligare två alternativ, `BUFFER` och `INOUT`.

En `BUFFER` skall användas för en utsignal som också kan ingå i uttryck för att tilldela andra signaler, dvs finns både i VL och HL. En `BUFFER` kan då vara både utsignal och mellanresultat. Skulle vi i fallet ovan använt `u` på det sättet hade vi måst ändra `PORT`-satsen till detta:

```
PORT(x, clk: IN bit; u: BUFFER bit);
```

En `INOUT`, till slut, används för dubbelriktade portar, när ett ben på kapseln skall växla mellan att vara in- och utgång. `INOUT` kan ersätta vilken som helst av de andra tre. Den är användbar för bl.a. databussar.

1.6 Lite grundläggande definitioner

I detta avsnitt skall vi gå lite närmare in på hur VDHL är definierat. Språket är icke case sensitive, dvs gör inte skillnad på stora och små bokstäver. `Begin`, `BEGIN` och `begin` är samma sak.

Kommentarer inleds med – (två minustecken). **Glöm inte bort att kommentera din kod!**

VHDL har stenhård typkontroll, så du kan inte låtsas att en integer är en Boolean, som man kan i C. För det mesta är detta av godo, håller koden tydlig och lättläst. Tyvärr har VHDL inte lika smidiga type casting-funktioner som t.ex. C, så det är knepigt att gå mellan typer när man behöver det.

1.6.1 Identifierare

När du namnger sina signaler så använder du, som i de flesta andra språk, identifierare. I VHDL gäller följande regler för namngivningen:

- En identifierare består av bokstäver, siffror och/eller understrykningstecken.
- Första tecknet måste vara en bokstav.
- Sista tecknet får inte vara understrykningstecken.
- Två understrykningstecken i följd får inte förekomma.
- Reserverade ord får inte användas som identifierare (t.ex. abs, architecture, begin, mm).

1.6.2 Signaler och variabler

VHDL skiljer på `SIGNAL` och `VARIABEL`. Vi undviker avsiktligt begreppet variabel. En variabel är bara definierad inom en process, och behöver inte egentligen finnas i den genererade konstruktionen.

I denna kurs rekommenderar vi att du inte krånglar till dina processer, utan håller dem enkla så att du har en hyfsad idé om vad de genererar för logik. Då har du inte heller någon större användning av variabler.

1.7 Datatyper

Hittills har vi bara tittat på konstruktioner som arbetar med en bit i taget. VHDL har stöd för datatyper som representerar flera signaler i en, som till exempel heltal uppbyggda av ett antal bitar, men också en bättre variabeltyp för enstaka bitar. Många av dessa typer är inte tillgängliga i den grund-VHDL som man har automatiskt, utan man måste utnyttja externa bibliiotek. I synnerhet kommer vi att ha nytta av biblioteket `ieee.std_logic_1164`.

Bättre på bitnivå: `std_logic`

Datatypen `bit`, den enda typ vi sett hittills, kan bara ha två värden: ’0’ och ’1’. Typen `boolean` är i princip samma sak, med värden `false` och `true`.

Även på bitnivå är detta ofta för begränsat. Det låter oss inte göra databussar, t.ex. för att använda RAM-minnen. För detta måste vi ha tillgång till tristate. Det får vi med `std_logic`, en datatyp som finns i biblioteket `std_logic_1164`. Det är en standardtyp som du skall använda i stället för `bit`. En signal av typen `std_logic` kan ha 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

`'U'`, `'X'` och är bara till för simulering. `'Z'` är för tristateutgångar och [’-’] kan man använda för att specificera ett näts uppförande (precis som i kursen Digitalteknik).

1.7.1 Numeriska datatyper

Integer fungerar ungefär som i C. Du kan deklarera en flerbitars signal som representerar ett heltal så här:

```
SIGNAL myinteger: integer RANGE 0 TO 7;
```

Detta heltal kommer rimligtvis att implementeras som tre vippor, dvs ta tre makroceller i anspråk. De mest uppenbara operationerna på numeriska datatyper är addition och subtraktion. De skrivs med + och −. Du kan också göra jämförelser, med >, < och =. Detta skrivs på normalt sätt, t.ex:

```
a <= b + c;
IF a > b THEN ...
```

Ett problem med integer är att du inte kan indexera den på bitnivå. Ofta vill du hellre använda `std_logic_vector`, som är en array som tillåter aritmetiska operationer. Se avsnittet om arrayer nedan.

VHDL stödjer också flyttal, bl.a. med den fördefinierade typen real, men dessa stödjs inte av alla syntesverktyg, och kräver i synnerhet mycket utrymme och är därför inte användbara i denna kurs.

1.7.2 Konstanter

Konstanter är som namnet säger fasta värden, som används för att underlätta ändringar och förtydliga koden. En konstant deklareras så här:

```
CONSTANT mynum: integer := 5;
```

Den deklareras alltså med både typ och värde.

1.7.3 Uppräkningsbara datatyper och egendefinierade datatyper

Du har också möjligheten att definiera dina egna typer, med `TYPE`. I den här kursens perspektiv är det mest intressant för uppräkningsbara datatyper. Dessa kan göra kod för ett sekvensnät mycket snygg och lättläst. Du definierar en typ med satsen `TYPE`. Så här ser den ut för en uppräkningsbar typ:

```
TYPE (typnamn) is (värde 1, värde 2... värde n-1, värde n);
```

Satsen `TYPE` kan användas för andra typer också, arrayer, records med mera, men det är utanför målen för denna kurs.

En typ som representerar tillstånden i ett sekvensnät kan se ut nånting i den här stilen:

```
TYPE tillstandstyp IS (ledig, ta_emot_data, error,
                       tagit_emot_data, vanta);
```

Sedan kan man deklarera signaler som använder typen:

```
SIGNAL nuvarande_tillstand, nasta_tillstand: tillstandstyp;
```

Det kan vara intressant att notera att den fördefinierade typen bit definieras så här:

```
TYPE bit is ('0', '1');
```

1.7.4 Grupperade signaler: array och record

En vektor (array) fungerar som i andra språk. Vi kommer i första hand att behöva arrayer av bitar.

Dessa kan lämpligtvis göras av typen `std_logic_vector`. Så här kan en array med fyra bitar deklareras:

```
SIGNAL myarray: std_logic_vector(3 downto 0);
```

Detta är en array bestående av bitarna `myarray(0)`, `myarray(1)`, `myarray(2)` och `myarray(3)`. Arrayer indexeras alltså med de vanliga parenteserna (), inte med hakparenteserna [] som man gör i de flesta vanliga språk.

Till skillnad från C så bestämmer du arraygränserna själv (inget krav på att de börjar på noll eller nåt annat dumt), så att du kan välja gränser så de passar ditt problem. Arraygränserna anges med `TO` eller `DOWNTO` på detta vis:

```
(undre gräns) TO (övre gräns)
```

eller

```
(övre gräns) DOWNTO (undre gräns).
```

En integer och en bitarray är i grund och botten samma sak, men VHDL låter oss inte göra saker som typerna inte är definierade för. När vi vill kunna jobba både aritmetiskt och på bitnivå så kan vi använda oss av typen `std_logic_vector`, som definieras i bilioteket `std_logic_unsigned`.

```
SIGNAL u: std_logic_vector(3 DOWNTO 0);
```

Nu kan vi göra allt detta med u:

```
u <= u + 1;
u <= "0101";
u(0) <= '1';
```

Vi rekommenderar att du alltid använder `std_logic` och `std_logic_vector` och inga andra datatyper. Minst signifikant bit är den med lägst nummer, bit 0 i exemplet ovan.

Man kan också gruppera signaler i records, strukturer (struct i C). Vi börjar med att definiera en egen datatyp, exempelvis:

```
TYPE controlword IS RECORD
   alu: std_logic_vector(3 DOWNTO 0);
   tobus: std_logic_vector(2 DOWNTO 0);
   halt: std_logic;
END RECORD;

SIGNAL styr1, styr2: controlword;
```

Tilldelningar går till så här:

```
styr1.halt <= '0';
styr1.alu <= "1011";
styr1.tobus <= styr2.tobus:
```

1.8 Mer kombinatorik

Följande satser kan vara användbara när du gör kombinatoriska nät. Dock bör du vara medveten om att detta gör implementationen ännu mindre självklar, och det har både för- och nackdelar.

1.8.1 Utanför `PROCESS`: `WITH-SELECT-WHEN`

Detta liknar case/switch i Algol-språken, men låt oss inte tänka i deras banor. Detta är i princip en multiplexer.

```
WITH (styrsignal) SELECT
  (utsignal) <= (uttryck 1) WHEN (signalvärde 1),
                (uttryck 2) WHEN (signalvärde 2),
                (uttryck 3) WHEN (signalvärde 3),
                ...
                (uttryck n-1) WHEN (signalvärde n-1),
                (uttryck n) WHEN others;
```

Detta gör att (utsignal) tilldelas något av (uttryck 1 till n) beroende på värdet av (styrsignal). `WHEN OTHERS` på slutet avslutar satsen.

Tag som exempel multiplexern i det tidigare exemplet på kombinatorik med Booleska uttryck:

```
y <= (not b AND not a AND c0) or (not b AND a AND c1) or
     (b AND not a AND c2) or (b AND a AND c3);
```

Låt oss i stället använda `WITH-SELECT-WHEN`. Vi ersätter `a` och `b` med vektorn `sel`: `signal sel: std_logic_vector(1 downto 0);` Då ersätts `a` av `sel(0)` och `b` av `sel(1)`. Muxen kan då uttryckas så här:

```
WITH sel SELECT
  y <= c0 WHEN "00",
       c1 WHEN "01",
       c2 WHEN "10",
       c3 WHEN others; -- 11
```

Ganska rakt på sak, eller hur?

1.8.2 Utanför PROCESS: WHEN–ELSE

Detta påminner om if-satsen i Algol-språk, men är annorlunda uttryckt för att påminna oss om att det är en parallell, omedelbar process snarare än ett test följt av utförande. Det är ett alternativt sätt att uttrycka sig när Booleska uttryck skulle bli bökiga.

```
(signal) <= (värde 1) WHEN (villkor 1) ELSE
            (värde 2) WHEN (villkor 2) ELSE
             ...
            (värde n-1) WHEN (villkor n-1) ELSE
            (värde n);
```

Satsen innehåller minst ett villkor och två värden att välja mellan, men kan alltså användas för ganska komplicerade uttryck. Se exempel i nästa avsnitt. Lägg märke till att `WITH-SELECT-WHEN`- och `WHEN-ELSE`-satsen innehåller exakt en signaltilldelning.

De liknande `CASE-WHEN` och `IF-THEN-ELSE` finns också, men används bara i processer, dvs sekvensnät, som behandlas i nästa avsnitt. Dessa innehåller en eller flera tilldelningar och är på så sätt mera kraftfulla.

1.9 Mer sekvensnät

Även för sekvensnät finns det lite mer avancerade styrsatser än vi hittills använt. De är minst lika användbara som de kombinatoriska satserna vi nyss gick igenom.

1.9.1 Inuti PROCESS: IF–THEN–ELSE

`IF`-satsen kan lite mer än vi sagt tidigare. En lite mer komplett form kan vara denna:

```
IF (uttryck 1) THEN
  (sats 1)
ELSIF (uttryck 2) THEN
  (sats 2)
  ...
```

```
ELSIF (uttryck n-1) THEN
  (sats n-1)
ELSE
  (sats n)
END IF;
```

Den minimala formen är den `IF-THEN-END IF` som vi sett tidigare, men vi kan alltså ha också ett godtyckligt antal `ELSIF` och ett eventuellt `ELSE`.

1.9.2 Inuti `PROCESS`: `CASE-WHEN`

En `CASE-WHEN`-sats är mycket lik case-satserna i Pascal:

```
CASE (styrsignal) IS
  WHEN (värde 1) => (sats 1);
  WHEN (värde 2) => (sats 2);
  ...
  WHEN (värde n-1) => (sats n-1);
  WHEN others => (sats n);
END CASE;
```

Symbolen => är en implikationspil. Tyvärr kan man ibland få den rätt förvirrande form som förekommer i exemplet nedan, där `=> d <=` pekar på signalen `d` från båda hållen, först en implikationspil och sen en tilldelningspil. Exemplet på sekvensnät ovan, sekvensnätet som detekterar sekvenser på tre eller fler ettor, kan man lätt skriva om så det använder både `IF` och `CASE`:

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

ENTITY trippel IS
  PORT(x, clk: in std_logic;
            u: out std_logic);
END trippel;

ARCHITECTURE two of trippel is
  SIGNAL q: std_logic_vector(0 to 1);
BEGIN

PROCESS (clk)
  BEGIN
  IF rising_edge(clk) THEN
    IF x = '0' THEN
      q <= "00";
```

```
    ELSE -- x är 1!
      CASE q IS
        WHEN "00" => q <= "01";
        WHEN "01" => q <= "11";
        WHEN "11" => q <= "10";
        WHEN others => q <= "10";
      END CASE;
    END IF;
  END IF;
END PROCESS;

u <= '1' WHEN q = "10" ELSE '0';
END two;
```

Detta sekvensnät fungerar precis likadant som det i avsnittet Sekvensnät med VHDL, men ser rätt annorlunda ut. Vi använder `CASE`-satsen för att på ett enkelt sätt stega fram genom tillstånden, så länge vi har insignalen `'1'`. En nolla skickar tillbaka oss till 00. Dessutom använder vi `IF` och `WHEN-ELSE`.

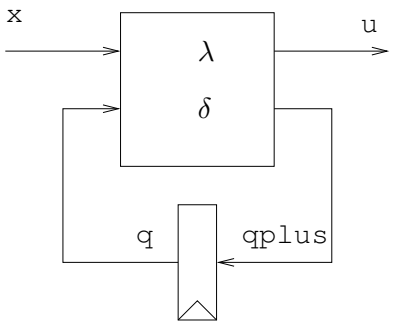
1.9.3 En utvidgning

Vi vill nu visa, som lite överkurs, ytterligare ett sätt att skriva ett sekvensnät. Vi tar som exempel samma nät som i föregående avsnitt. Eftersom `ENTITY`-satsen är den samma, så hoppar vi över den och går direkt på `ARCHITECTURE`-satsen:

```
ARCHITECTURE three of trippel is
  SIGNAL q,qplus: std_logic_vector(0 to 1);
BEGIN

-- tillståndsvipporna
PROCESS (clk)
  BEGIN
  IF rising_edge(clk) THEN
    q <= qplus;
  END IF;
END PROCESS;

-- delta- och lambda-näten
PROCESS (q,x)
  BEGIN
    qplus <= "00"; -- defaultvärden
    u <= '0';
    CASE q IS
      WHEN "00" =>
        u <= '0';  -- Moore!
```

```
        IF x='1' THEN
          qplus <= "01";
        END IF;
      WHEN "01" =>
        u <= '0';
        qplus <= "11";
      WHEN "11" =>
        u <= '0';
        IF x='1' THEN
          qplus <= "10";
        END IF;
      WHEN others =>
        u = '1';
        IF x='1' THEN
          qplus <= "10";
        END IF;
    END CASE;
END PROCESS;

END three;
```

Vad har vi gjort? Vi börjar med lite reverse engineering och ritar upp ett blockschema, se figur 1.9. Precis som förut har vi två block, men vi har gjort en annan uppdel-



Figur 1.9: Sekvensnät med hopslagna δ- och λ-nät.

ning. Vi har gjort ett block, en klockad process, av endast tillståndsvipporna. För att göra det har vi fått införa ett nytt signalnamn, `qplus`, för nästa tillstånd.

All kombinatorik, dvs δ- och λ-näten har samlats i en kombinatorisk process. Fördelen med detta är förstås att vi nu får nästa-tillstånd- och ut-signalen på samma ställe. Att skriva inkråmet i denna process känns precis som att rita tillståndsgrafen. Nackdelen är att den kombinatoriska processen ställer lite större krav på förståelse och måste vara, just det, kombinatorisk. Följande gäller:

- alla insignaler till blocket måste räknas upp i sensitivity-listan.

- klockan får inte förekomma någonstans i blocket.
- alla insignalfall måste täckas för att undvika att få oönskade latchar. Alla `CASE`- och `IF`-satser måste vara fullständiga. För att få mindre att skriva rekommenderar vi att sätta default-värden först i blocket och bara ta med de fall då vi inte ska ha default-värdena. Gör så och slipp latchar!

Vi avslutar detta avsnitt med en förmaning. Är du det minsta osäker på skillnaden mellan klockade och kombinatoriska processer, så undvik de sistnämnda.

1.10 Lite extra för att prata med omvärlden

I detta avsnitt tar vi upp ett par detaljer som behövs för att prata med omvärlden, dels hur VHDL importerar från bibliotek, och dels hur du specifierar konstruktionens destination.

1.10.1 Att använda externa bibliotek

VHDL har, likt de flesta moderna programmeringsspråk1, direkt stöd för att komma åt typer och kod i andra moduler. I VHDL används det reserverade ordet `USE`.

Du kommer inte att behöva använda så många andra moduler/bibliotek i Digital Konstruktion, men i mer avancerade VHDL-kurser kommer det att användas desto mer. Här behöver vi oftast bara använda biblioteket `ieee.std_logic_1164`. Följande rader rekommenderas i början av dina program (före `ENTITY`):

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

Det du vill ha ur dessa bibliotek är ett antal standardtyper som definierar normala in- och utsignaler, som `std_logic` och `std_logic_vector`.

1.11 Syntetiserbar VHDL

All VHDL-kod är inte syntetiserbar! Det finns många konstruktioner som är korrekt VHDL-kod men som inte kan realiseras. Tag som exempel följande felaktiga D-vippa:

```
PROCESS (clk)
BEGIN
  IF clk'event THEN
    q <= d;
  END IF;
END PROCESS;
```

Konstruktionen `clk'event` har inte nämnts tidigare i detta kompendium. Man säger att `'event` är ett attribut till signalen `clk`. Uttrycket `clk'event` är sant när signalen `clk` har ändrats.

Denna vippa ser ut att reagera på varje förändring av klockan, både uppåt- och neråtflank. Du kan inte räkna med att sådana vippor finns i en normal programmerbar krets. Om du testar koden i ISE kommer resultatet faktiskt att bli oberoende av klockan, bara en rak vidarekoppling med en liten fördröjning.

Din kod kan också vara korrekt även för den krets du använder, men helt enkelt för stor så att den inte får plats. I det fallet får du dock felmeddelande från kompilatorn.

Kapitel 2

Utvecklingsverktyget ISE

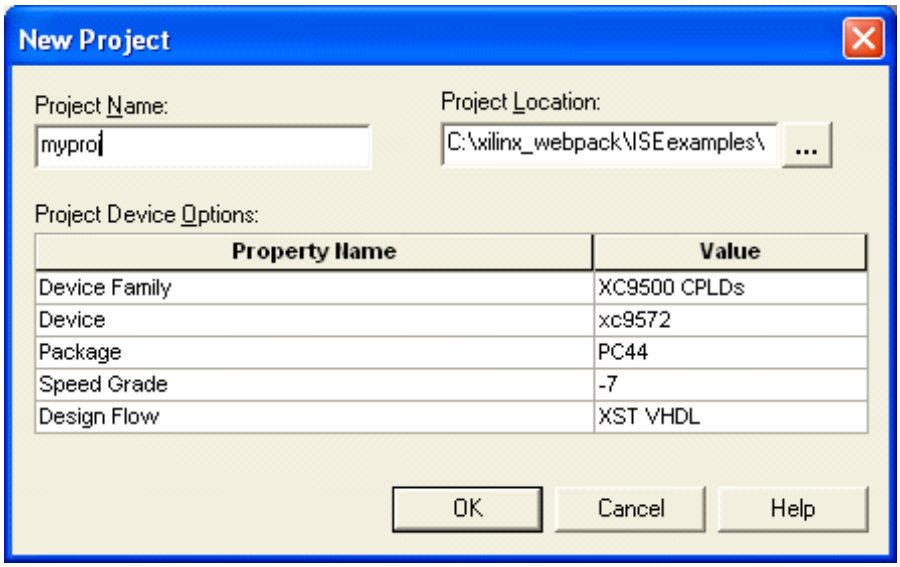
Vi kommer i denna kurs att använda ISE från Xilinx. ISE består av två integrerade delar

- **Project Navigator**. Här kompilerar och syntetiserar du dina VHDL-filer.
- **ModelSim**. En mycket kompetent VHDL-simulator. ModelSim kan också användas helt fristående. Vi tycker dock att det är enklare att anropa den från Project Navigator.

ISE finns även i en gratisvariant, Webpack, som kan laddas ner gratis från Xilinx hemsida. Sök med Google efter "xilinx webpack". Missa inte ModelSim i installationen.

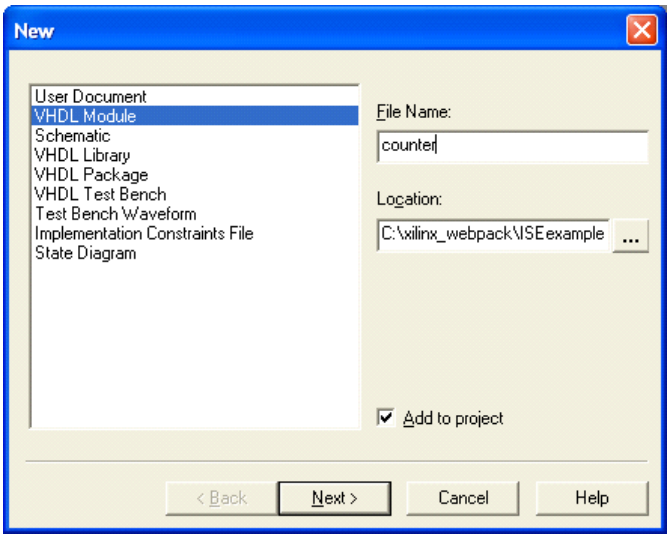
2.1 En snabblektion i Project Navigator

Börja med att klicka på ikonen **Project Navigator**. Välj sedan **File->New Project**. Fyll i fönstret New Project på följande sätt:

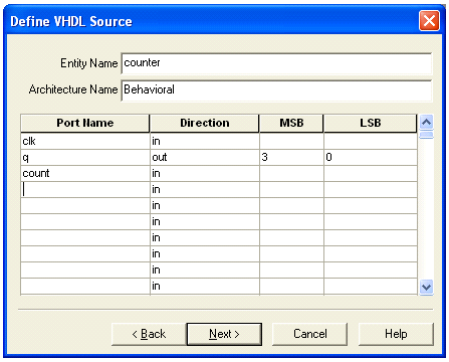


Här anges alltså vilken typ av PLD, som vi ska använda. Dessutom måste vi tala om att det är ett VHDL-projekt vi tänker satsa på.

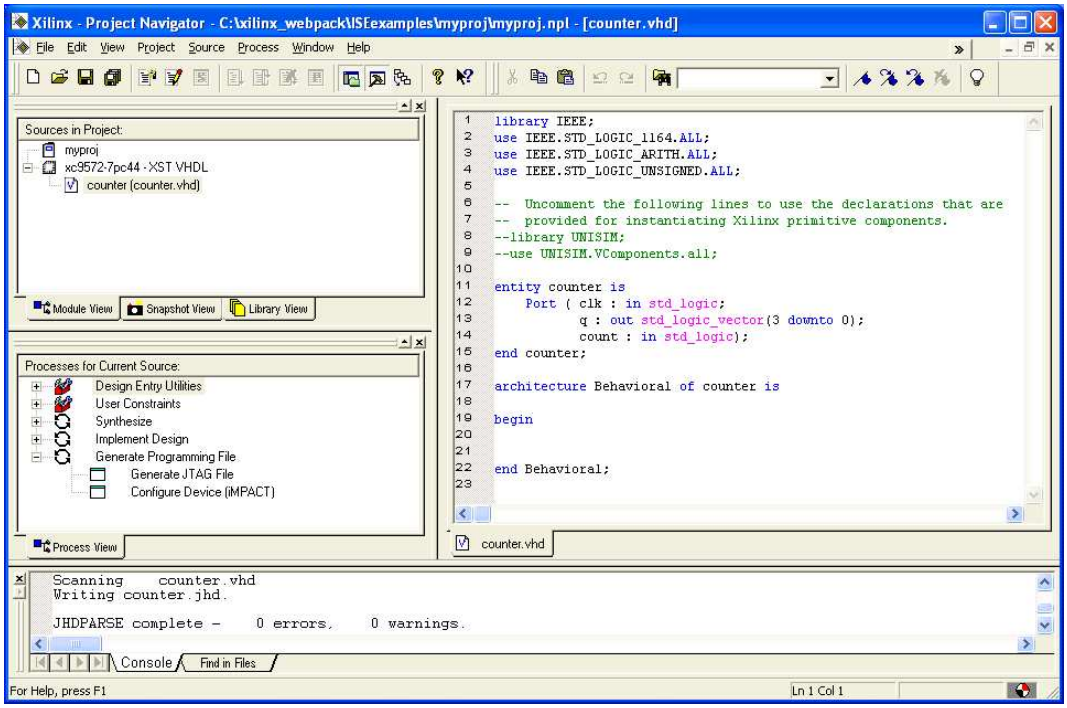
Nu är det dags att fylla på lite VHDL-kod. Välj **Project->New Source**.



Vi väljer VHDL module och ett namn på File name. Tryck därefter **next**. Här väljer vi in- och utgångar.



Tryck därefter **next** och sedan på **Finish**. Har allt gått som det ska, ser vi då detta fönster:



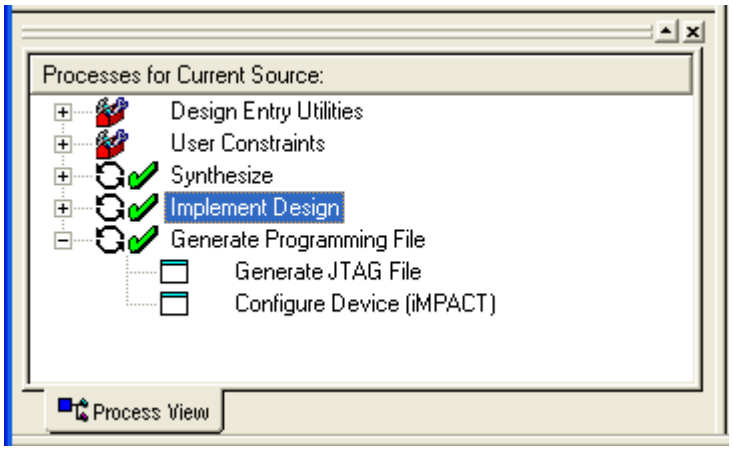
Som synes har ISE gett oss ett kodskelett. Vi fyller i VHDL-koden för en enkel räknare. Vi nöjer oss med följande minimala `ARCHITECTURE`.

```
architecture Behavioral of counter is
begin
process(clk)
begin
  if rising_edge(clk) then
    if count='1' then
      q <= q+1;
```

```
      end if;
    end if;
  end process;
  end Behavioral;
```

Eftersom vi läser av `q`, finns i HL, måste vi ändra moden till `BUFFER`. Det kunde ju inte ISE veta när vi fyllde i tabellen.

Dubbelklicka nu i tur och ordning på **Synthesize**, **Implement Design** och **Generate Programming File**. Går allt väl tänds tre gröna bockar, se figur 2.1, vid respektive rad, om inte finns felutskrifter att läsa i console-fönstret.



Figur 2.1: Så här ska det se ut, tre gröna bockar!

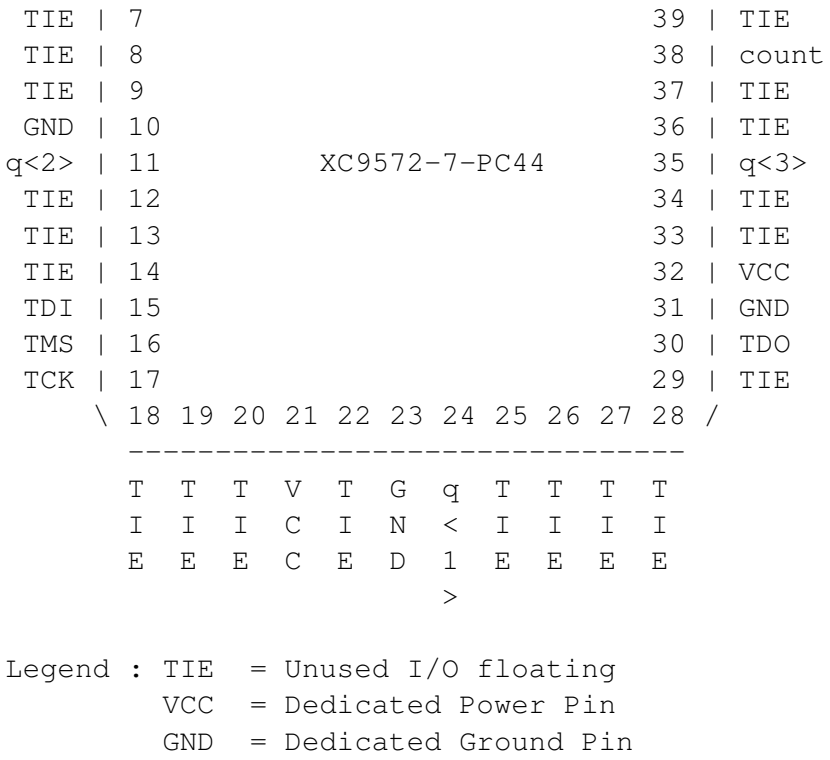
Dubbelklicka på **Implement Design->Fit->Fitter Report**. Ur rapporten saxar vi följande information:

```
Macrocells     Product Terms   Registers      Pins           Function Block
Used           Used            Used           Used           Inputs Used
4 /72(6%)      4/360(1%)       4/72(6%)       6/34(18%)      10/144  (7%)
```

När en krets blir full är det en av ovanstående resurser, som tagit slut. Här är det som synes ingen fara!

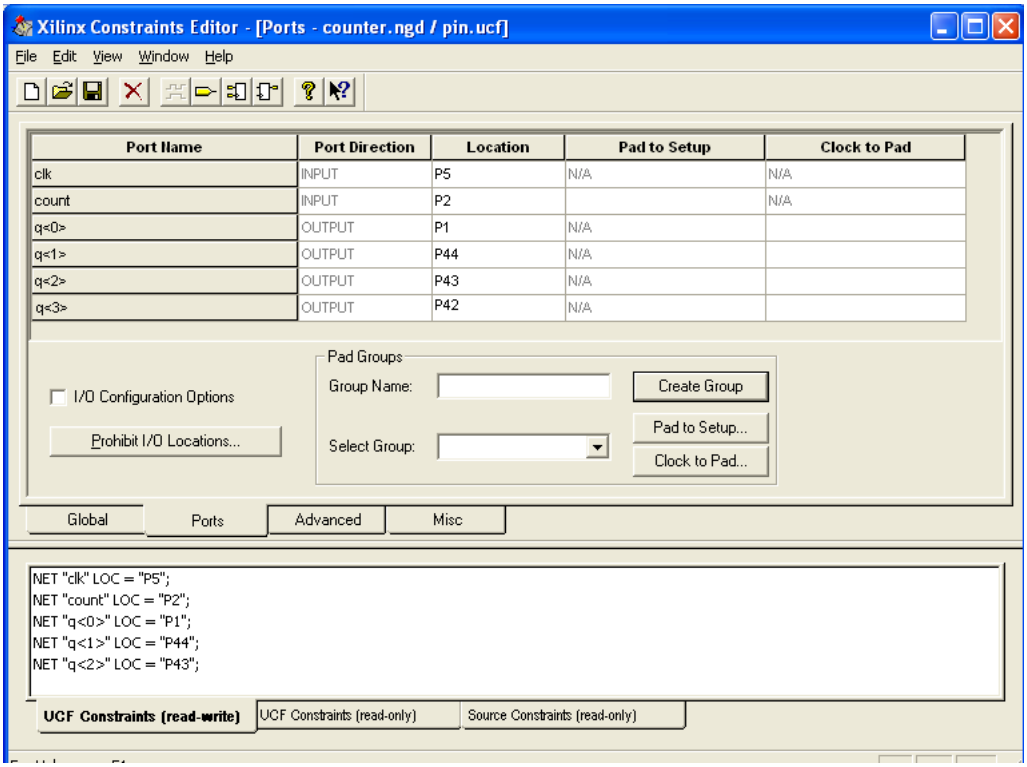
Ur samma rapport (längre ner) hittar vi också Device Pin Out, alltså på vilka pinnar våra signaler har hamnat.

```
                        q
    T   c   T   T   T   <   T   T   T   V   T
    I   l   I   I   I   0   I   I   I   C   I
    E   k   E   E   E   >   E   E   E   C   E
    ------------------------------------
  /6   5   4   3   2   1   44  43  42  41  40 \
```




```
Legend : TIE  = Unused I/O floating
         VCC  = Dedicated Power Pin
         GND  = Dedicated Ground Pin
```

Vi är dock inte nöjda med denna benplacering, utan vill låsa signalerna till vissa pinnar. Gör **Project->New Source**, välj **Implementation Constraints File** och namnge filen, t ex **pin**. I rutan **Sources in Project** syns nu en ny fil: **pin.ucf**. Dubbelklicka på den och välj fliken **Ports**! Vi gör följande val: `clk` får ligga kvar på pinne 5 (det är ingen slump att den hamnat här), `count` pinne 2 och `q` på pinnarna 1,44,43,42. Alla pinnar är förstås inte I/O-pinnar, utan endast de som är märkta med `TIE` i benplaceringen är tillgängliga för oss.



Eftersom vi har ändrat på benplaceringen måste vi köra om kompileringen. Klicka på file `counter.vhd` och dubbelklicka sedan på de gula frågetecknen, som tänts istället för de gröna bockarna. Nu skulle vi kunna programmera en krets genom att dubbelklicka på **Configure Device (iMPACT)** och sedan följa anvisningarna (förutsatt att ett programmeringskort är anslutet till just vår maskin). Vi skjuter dock upp programmeringen till avsnittet 3.6 Programmering av CPLD. Vi går istället vidare till att simulera vår kod.

2.2 Simulering i ModelSim

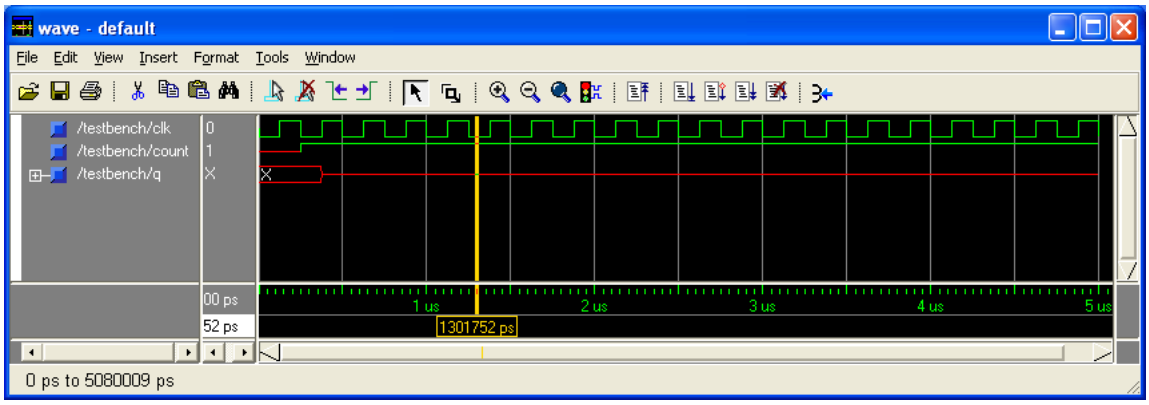
ModelSim är en VHDL-simulator. ModelSim simulerar alltså VHDL-koden direkt och är inte beroende av val av CPLD. Även ej syntetiserbar kod går att simulera.

Börja med att göra, fortfarande i Project Navigator, **Project->New Source** och välj **VHDL Testbench** och ett namn. Vi får nu ett kodskelett, som vi kompletterar på följande sätt:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
```

```
END testbench;

ARCHITECTURE behavior OF testbench IS
        COMPONENT counter
        PORT(clk : IN std_logic;
             count : IN std_logic;
             q : BUFFER std_logic_vector(3 downto 0));
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL count :  std_logic;
        SIGNAL q :  std_logic_vector(3 downto 0);

BEGIN
        uut: counter PORT MAP(
                clk => clk,
                count => count,
                q => q);

-- ********** lägg till här ************
-- klockan
   PROCESS
   BEGIN
      clk <= '0';
      wait for 125 ns;
      clk <= '1';
      wait for 125 ns;
   END PROCESS;
-- övriga signaler
   count <= '1';
END;
```

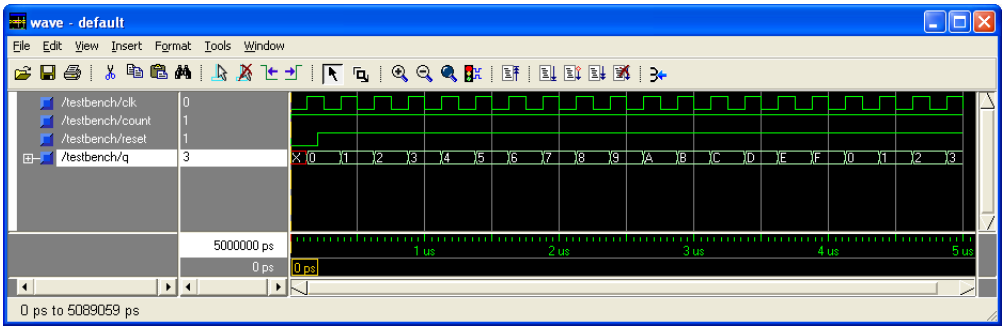
Vi har alltså gjort en klocka på 4 MHz och låter signalen `count` vara aktiv hela tiden. Vi högerklickar på **Simulate Behavioral VHDL Model** och ställer upp simuleringstiden till 5 μs. Därefter dubbelklickar vi på **Simulate Behavioral VHDL Model**. ModelSim startar med många fönster, varav vi koncentrerar oss på detta:



Här upptäcker vi ett fel. Signalen `q` har inget startvärde. Efter lite grunnande beslutar vi oss föra att införa en `reset`-signal, synkron och aktivt låg. Det innebär omskrivningar både i `counter` och testbänken. Vi använder bl a följande rad för att göra en `reset`-signal i testbänken.

```
reset <= '0', '1' after 200 ns;
```

Nu blir simuleringsresultatet så här:



Lägg märke till att tidsfördröjningarna är exempel på ej syntetiserbar VHDL. Du hittar ytterligare ett exempel på en testbänk i bilaga B.2.

2.2.1 COMPONENT-satsen

Med `COMPONENT`-satsen kan du alltså använda en VHDL-`ENTITY/ARCHITECTURE` som en komponent i en överordnad VHDL-`ENTITY/ARCHITECTURE`. VHDL är sin vana troget ganska pratigt. Du måste

- *deklarera* komponenten mellan `ARCHITECTURE` och `BEGIN` tillsammans med de andra deklarationerna.
- *instantiera* komponenten mellan `BEGIN` och `END`. Dessutom måste du koppla in komponenten. Detta sker med satsen

```
uut: counter PORT MAP(
        clk => clk,
        count => count,
        q => q
);
```

Pilarna betyder inte signalriktning utan signal inuti => signal utanför komponenten, alltså *kopplas till*. I detta fall (bestämt av Project Navigator) har dessa signaler samma namn.

Vi tror att du ska bara använda COMPONENT-satsen i testbänkar (och då får ju du den gratis av verktyget). CPLD 9572 är för liten för sk hierarkisk konstruktion.

Ett annat användningsområde är om du vill simulera ett bygge, som består av flera CPLD-er.

Kapitel 3

Ett miniprojekt: Personnummerkontroll

3.1 Inledning

Vi kommer här att gå igenom ett projekt, precis som du kommer att göra i kursen. Innan vi drar igång, ett litet allvarsord:

1. Tänk först hårdvara och gör ett blockschema.
2. Översätt sedan till VHDL, block för block.

Skriv aldrig kod först för att sedan hamna i diskussioner om vad denna kod egentligen gör!

3.2 Kravspecifikation

Bakgrund

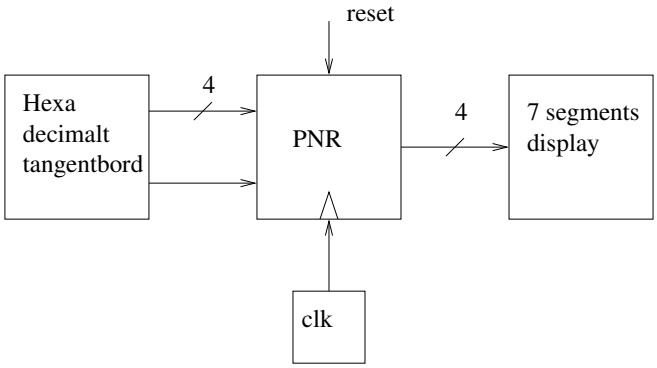
Ett tangentbord med siffrorna 0-9 används för att mata in de nio första sifforna i ett svenskt personnummer. Den sista siffran, som bara är en slags checksumma, skall genereras automatiskt när ytterligare en knapp trycks ner. Siffrorna skall dyka upp en och en på en display samtidigt som de matas in. Den sista siffran d_{10} bildas ur de första nio $d_1d_2d_3d_4d_5d_6d_7d_8d_9$ så att följande uttryck gäller

$$2d_1 + d_2 + 2d_3 + d_4 + 2d_5 + d_6 + 2d_7 + d_8 + 2d_9 + d_{10} + n \equiv 0 \pmod{10},$$

dvs vänsterledet ska vara ett jämnt tiotal. Talet n anger hur många av multiplikationerna med 2 som genererade en tiotalssiffra.

Grovt blockschema

Vi börjar med att rita ett grovt blockschema, som egentligen bara är en illustration av `ENTITY`-satsen.



Kravlista

Ska-krav

1. Siffrorna ska matas in från ett hexadecimalt tangentbord.
2. Siffrorna ska visas en i taget på en 7-segments display.
3. Efter 9 inmatade siffror ska kontrollsifran visas när en godtycklig knapp trycks ner.
4. Konstruktionen ska vara redo för ett nytt personnummer efter att föregående kontrollsiffra visats.
5. Exakt en XC9572 ska ingå i konstruktionen.
6. En reset-knapp ska finnas. Denna ska nollställa beräkningen, så att första siffran kan knappas in.

Bör-krav

1. Det bör vara möjligt att backa en siffra i personnumret.

3.3 Designskiss

Analys av problemet

Uppgiften består alltså av att beräkna

$$S_9 = (\sum_{k=0}^{8} w_k d_k + n_k) \mod 10$$
$$d_9 = (10 - S_9) \mod 10,$$

där w_k är 2 om k är udda och 1 annars. n_k är 1 eller 0 beroende på om $w_k d_k > 9$ eller inte. I fortsättningen låter vi alla plus/minus-tecken vara modulo 10 och satsar på en ackumulatorbaserad algoritm:

$$\begin{aligned} S_0 &= 0 \\ S_{k+1} &= S_k + \underbrace{w_k d_k + n_k}_{X2(d_k,k)}, \quad k = 0 \ldots 8 \\ d_9 &= 10 - S_9, \end{aligned} \tag{3.1}$$

där X2 är ett separat kombinatoriskt nät. Vår förhoppning är att all aritmetik ska utföras modulo 10, dvs med 4 bitar.

Förfinat blockschema med förklaringar

Nu ska vi alltså tänka hårdvara! Eftersom vi redan gjort en del matematisk analys av problemet, så kan vi nästan direkt översätta ekvation (3.1) till ett förfinat blockschema, vars centrum är ackumulatorregistret, se figur 3.1. Självklart konstruerar vi synkront, dvs alla vippor klockas samtidigt. Dessutom laddar vi de två registren samtidigt, dvs

$$\begin{aligned} d_k &= \text{KB} \\ S_{k+1} &= S_k + X2(d_k, k) \end{aligned}$$

utförs samtidigt.

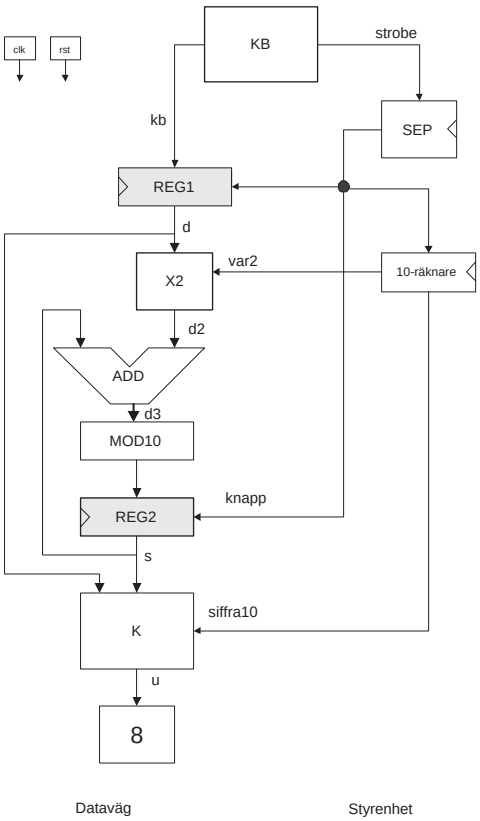
Här följer en kortfattad beskrivning av varje blocks funktion.

SEP : Hit kopplas stroben från tangentbord och synkronisering och enpulsning utförs.

REG1 : 4-bitars register, som håller d_k. Detta register är faktiskt inte nödvändigt, men konstruktionen blir trevligare att simulera och felsöka eftersom siffran ligger kvar till nästa tryckning.

10-räknare : Vanlig dekadräknare, vars LSB (`var2`) styr nätet `X2`. Signalen `siffra10` indikerar att räknaren har nått sitt ändläge.

X2 : Kombinatoriskt nät som bestäms av följande sanningstabell:



Figur 3.1: Förfinat blockschema för personnummermaskinen. Lägg märke till den typiska uppdelningen i en dataväg till vänster och en styrenhet (SEP och 10-räknare) till höger. Registren i datavägen har för tydlighetens skull gjorts gråa.

X2		
in	ut (var2=0)	ut (var2=1)
0	0	0
1	1	2
2	2	4
3	3	6
4	4	8
5	5	1
6	6	3
7	7	5
8	8	7
9	9	9

ADD : Vanlig 4-bitars adderare med 5 bitars summa.

MOD10 : Binär-BCD-omvandling.

REG2 : Detta är ackumulatorregistret, som håller värdet S_k. På ingången finns S_{k+1}.

K : Kombinatoriskt nät som muxar fram inknappad siffra eller kontrollsiffra beroende på signalen `siffra10`. Kontrollsiffran beräknas genom formeln $u = (10 - s) \mod 10$.

Övrigt

- Antalet vippor kan faktiskt redan nu bestämmas: 14 st. (4 vardera i REG1,2 och 10-räknare, 2 i SEP). Vi tror därför att konstruktionen ska gå in i en 9572.
- Klockfrekvensen väljes till 1 MHz. Klockfrekvensen bör vara högre än 100 Hz för att inte missa stroben och lägre än 10 Mhz pga virtekniken.

3.4 Översättning till VHDL

Nu återstår bara att, block för block, översätta vår lösning i figuren 3.1 till VHDL.

3.4.1 Synkronisering och enpulsning

Vi synkroniserar och enpulsar mha två D-vippor. Observera att tilldelningsatsen med `knapp` ska vara utanför processen, annars får vi tre vippor.

```
-- SEP synk och enpulsning av strobe
process(clk,rst)
begin
  if rst='0' then
    x <= '0';
    y <= '0';
  elsif rising_edge(clk) then
    x <= stb;
    y <= x;
  end if;
end process;

knapp <= x and (not y);
```

3.4.2 10-räknare

En dekadräknare, där vi bara använder utsignalerna `var2` och `siffra10`.

```
--  10-räknare
process(clk,rst)
begin
  if rst='0' then
```

```
    p <= "0001";
  elsif rising_edge(clk) then
    if knapp='1' then
      if p<10  then
        p <= p+1;
      else
        p <= "0000";
      end if;
    end if;
  end if;
end process;

var2 <= p(0);
siffra10 <= '1' when p=10 else '0';
```

3.4.3 Register

Ett enkelt register som laddas med signalen `knapp`.

```
-- REG1
process(clk,rst)
begin
  if rst='0' then
    d <= "0000";
  elsif (rising_edge(clk) and knapp='1') then
    d <= kb;
  end if;
end process;
```

3.4.4 X2

Observera att `var2=1` i alla `else`-grenar utom den första.

```
-- X2
d2 <= d when var2='0' else
      "0000" when d=0 else
      "0010" when d=1 else
      "0100" when d=2 else
      "0110" when d=3 else
      "1000" when d=4 else
      "0001" when d=5 else
      "0011" when d=6 else
      "0101" when d=7 else
      "0111" when d=8 else
      "1001" when d=9 else
```

```
      "0000";
```

3.4.5 Adderare

Observera att `d3` är 5 bitar och `s, d2` är 4 bitar.

```
-- ADD
d3 <= ('0' & s) + ('0' & d2);
```

3.4.6 Register med mod-10 på ingången

```
-- REG2 och MOD10
process(clk,rst)
begin
  if rst='0' then
    s <= "0000";
  elsif rising_edge(clk) then
    if knapp='1' then
      if d3<10 then
        s <= d3(3 downto 0);
      else
        s <= d3(3 downto 0) + 6;
      end if;
    end if;
  end if;
end process;
```

3.4.7 Nätet K

```
-- K
u <= d when siffra10='0' else
     "0000" when s=0 else
     10-s;
```

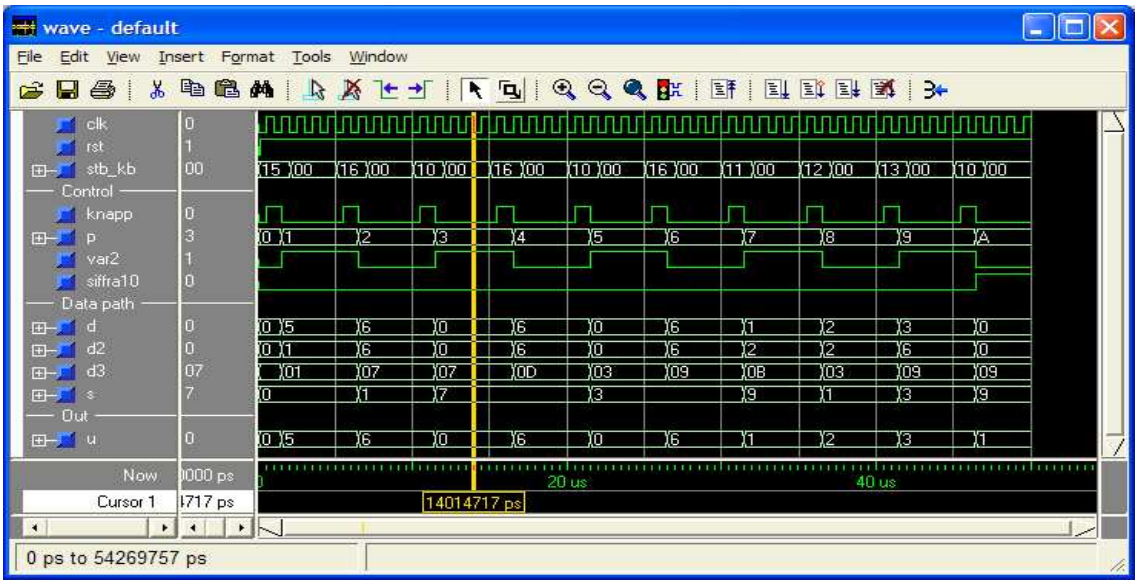
3.4.8 Kommentar

Vi är nu ganska nöjda med vår konstruktion. Vi vill särskilt påpeka att

- vi nu har vår design representerad på tre olika sätt. Matematiska formler, förfinat blockschema och VHDL-kod.
- vår kod består av ganska många små och lättförståeliga processer. Eftersom vi har kommit fram till koden via ett hårdvaruschema, så uppstår inte problem med multipel tilldelning. Varje signal tilldelas ett värde på exakt ett enda ställe.

3.5 Simulering

Vi avslutar detta exempel med en simulering av personnummermaskinen, se figur 3.2. Du hittar komplett VHDL-kod för personnummermaskinen i bilaga B.1 och en testbänk i bilaga B.2. I figuren 3.3 beskrivs hur plotten i figur 3.2 byggs upp.



Figur 3.2: Simulering av personnummermaskinen. Så här snygg går det alltså att få plotten. Vi simulerar naturligtvis alla intressanta signaler. Det går att sätta egna namn på signalerna.

Vid simuleringen vill du naturligtvis titta på alla signalerna i blockschemat i figuren 3.1. Det är ju med detta blockschema vi har tänkt fram vår lösning och vi har en föreställning om vad signalerna bör vara efter varje knapptryckning. I figurtexterna till figur 3.2 och 3.3 finns en förklaring på hur det går till.

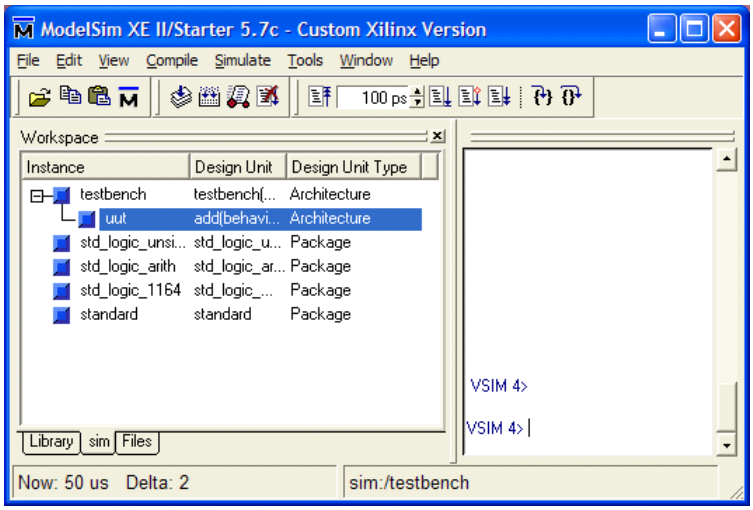
Glöm inte att spara utseendet på wave-fönstret med **File->Save Format...**. Du kan köra en simulering med kommandot **run 50 us** på kommandoraden Modelsims huvudfönster, se figur 3.3. Omstart, utan frågefönster, sker med **restart -f**.

3.6 Programmering av CPLD 9572

1. Resultatet av syntetisering blir en fil med suffixet .jed. Flytta den filen till en brännar-PC.

2. Montera en 9572-a i hållaren, som visas i figur 3.4.

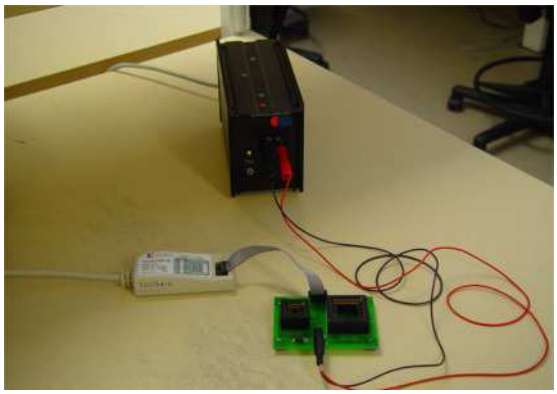
3. Se till att nätaggregatet är påslaget.



Figur 3.3: Modelsims huvudfönster. Till vänster finns fliken sim. Högerklicka på uut (så heter faktiskt vår instantierade konstruktion) och välj View Source. I textfönstret kan du nu dubbelklicka på önskad signal och dra den till wave-fönstret.

4. Starta programmet iMPACT och svara på frågorna. Låt programmet själv upptäcka vilken krets som sitter i hållaren mha **boundary scan**.

5. Högerklicka på kretsen i fönstret och välj **program**.



Figur 3.4: Programmeringssockel för 9572.

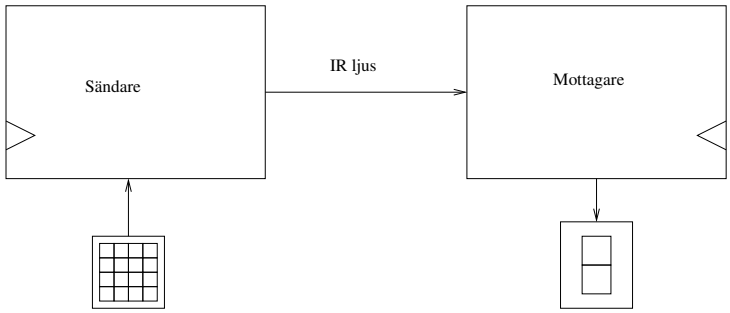
Kapitel 4

Laborationsuppgift: IR-mottagare

Laborationen är framtagen av Lennart Bengtsson och används också i kursen Elektronikprojekt Y.

4.1 Inledning

En siffra inmatad från ett hexadecimalt tangentbord ska sändas seriellt över en IR-länk och presenteras på mottagarsidan på en sjusegmentsdisplay. Sändaren kommer



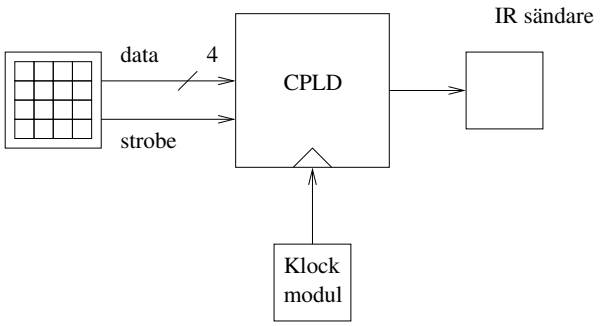
Figur 4.1: En IR-överföring, bestående av sändare och mottagare.

att tillhandahållas vid laborationstillfället. Din uppgift blir att konstruera mottagaren.Observera att sändaren och mottagaren inte har samma klocka. Frekvensen, däremot, bör vara så lika som möjligt.

4.2 Sändaren

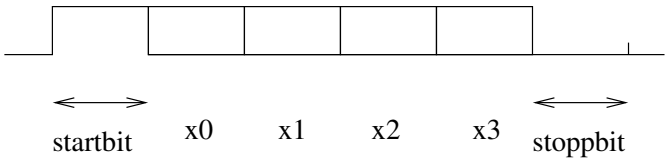
Denna modul finns färdig på labplatsen.

Sändaren utgörs av en CPLD, ett hexadecimalt tangentbord, en klockmodul och en IR-sändare, se figur 4.2. För varje knappnedtryckning sänds en 16 klockintervall



Figur 4.2: IR-sändare.

lång startbit (alltid 1), och 4 bitar, också de 16 klockintervall långa, se figur 4.3. LSB sänds först. Under pågående sändning får inga tryckningar göras. Efter den sista biten är ledningen 0 minst 16 klockintervall. Denna 0:a ska användas som en stoppbit.



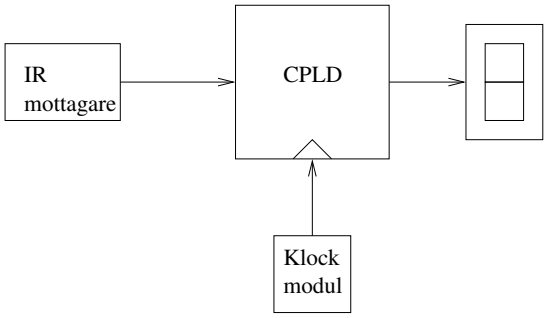
Figur 4.3: Pulståg från sändaren.Alla bitar är 16 klockintervall långa.

De ingående komponenterna fungerar på följande sätt:

- *hexadecimalt tangentbord*: En knappnedtryckning medför att data $< x_3x_2x_1x_0 >$ läggs ut på datautgångarna. När data ligger stabilt aktiveras signalen strobe (aktivt hög). När knappen släpps upp går strobe låg, men data ligger kvar.
- *IR-sändaren*: Sänder en modulerad signal så länge dess insignal är logiskt ett.
- *Klockmodul*: Frekvensen är justerbar i området 1-1000 Hz. Vid laborationen kan du anta att klockfrekvensen är 1000 Hz, dvs varje bit är 16 ms lång och hela tecknet 96 ms.

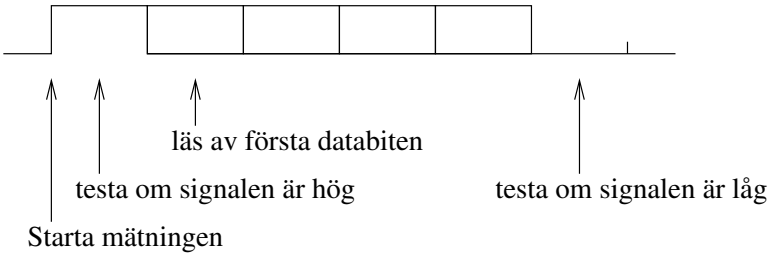
4.3 Mottagaren

Mottagaren ska utgöras av en CPLD (XC9572), en BCD/sjusegmentsavkodare med tillhörande display, en klockmodul och en IR-mottagre. Mottagningen ska starta så snart en positiv puls har detekterats på ingången. Efter 8 klockpulser, dvs på mitten av förmodad startbit ska ett nytt test utföras. Visar testet att insignalen nu återgått till noll, är detta att betrakta som störning och inget data följer. Om insignalen



Figur 4.4: IR-mottagare.

fortfarande är ett, är det en giltig startbit som följs av databitar. Dessa ska läsas av så nära mitten som möjligt i intervallen, så att en viss avvikelse mellan klockfrekvenserna kan tillåtas. Efter att en siffra tagits emot ska mottagaren vara redo för nästa siffra. Observera att det är viktigt att detektera stoppbiten innan testet för nästa startbit börjar. Stoppbiten är nödvändig för att garanterat hitta nästa startbit. Mottagaren ska klara av att ta emot flera meddelanden i följd utan att behöva resettas.



Figur 4.5: Mottaget pulståg.

De ingående komponenterna fungerar på följande sätt:

- *IR-mottagaren*: Insignalen demoduleras och en etta genereras så länge insignal finns, annars en nolla.
- *Klockmodul*: Frekvensen är justerbar i området 1-1000 Hz.

4.4 Uppgifter

1. Börja med att göra ett blockschema i stil med figur 3.1. Även här är det fruktbart att dela konstruktionen i en dataväg och en styrenhet. Ledtråd: använd ett skiftregister i datavägen. Styrenheten kan innehålla en räknare och ett sekvensnät.
2. Översätt blockschemat till VHDL.

3. Skriv en testbänk och simulera. Titta på samtliga signaler, inte bara signalerna i `ENTITY`-satsen. I testbänken ska du simulera insignalen i figuren 4.5. Gör en klocka på 1 kHz, en lämplig reset-signal och den mottagna IR-signalen

```
-- startbit + 4 bitar + 1 stoppbit, dvs 96 CP
IR <= '0', '1' after 2.5 ms, '0' after 82.5 ms,
           '1' after 98.5 ms, '0' after 114.5 ms;
```

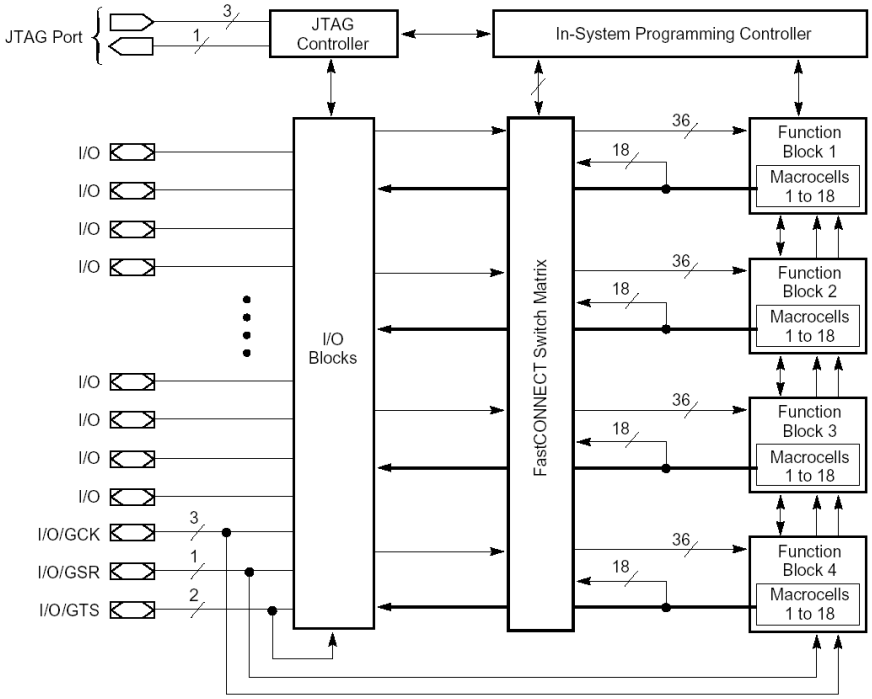
   Här skickade vi alltså två meddelanden 1111 och 0000 direkt efter varandra för att testa att vi hanterar stoppbiten rätt.

4. Programmera en krets, koppla upp och provkör.

Bilaga A

Kopplingsscheman

A.1 Blockschema 9572



A.2 Pinnar 9572

XC9572 Global, JTAG and Power Pins

Pin Type	PC44	PC84	PQ100	TQ100
I/O/GCK1	5	9	24	22
I/O/GCK2	6	10	25	23
I/O/GCK3	7	12	29	27
I/O/GTS1	42	76	5	3
I/O/GTS2	40	77	6	4
I/O/GSR	39	74	1	99
TCK	17	30	50	48
TDI	15	28	47	45
TDO	30	59	85	83
TMS	16	29	49	47
V_{CCINT} 5 V	21,41	38,73,78	7,59,100	5,57,98
V_{CCIO} 3.3 V/5 V	32	22,64	28,40,53,90	26,38,51,88
GND	10,23,31	8,16,27,42, 49,60	2,23,33,46,64,71, 77,86	100,21,31,44,62,69, 75, 84
No Connects	–	—	4,9,21,26,36,45,48, 75, 82	2,7,19,24,34,43,46, 73, 80

Bilaga B

VHDL-kod

B.1 Komplett VHDL-kod för personnummermaskinen

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

entity add is
  Port ( clk,rst, stb: in std_logic;
         kb : in std_logic_vector(3 downto 0);
         u : out std_logic_vector(3 downto 0));
end add;

architecture Behavioral of add is
  signal x,y,knapp,var2,siffra10: std_logic;
  signal p:  std_logic_vector(3 downto 0);
  signal d, d2:   std_logic_vector(3 downto 0);
  signal d3:    std_logic_vector(4 downto 0);
  signal s:   std_logic_vector(3 downto 0);
begin
  -- SEP sync and single pulse strobe
  process(clk,rst)
  begin
    if rst='0' then
      x <= '0';
      y <= '0';
    elsif rising_edge(clk) then
      x <= stb;
      y <= x;
    end if;
  end process;
```

```
knapp <= x and (not y);

--  10-räknare
process(clk,rst)
begin
  if rst='0' then
    p <= "0000";
  elsif rising_edge(clk) then
    if knapp='1' then
      if p<10  then
        p <= p+1;
      else
        p <= "0001";
      end if;
    end if;
  end if;
end process;
var2 <= p(0);
siffra10 <= '1' when p=10 else '0';

-- REG1
process(clk,rst)
begin
  if rst='0' then
    d <= "0000";
  elsif (rising_edge(clk) and knapp='1') then
    d <= kb;
  end if;
end process;

-- X2
d2 <= d when var2='0' else
      "0000" when d=0 else
      "0010" when d=1 else
      "0100" when d=2 else
      "0110" when d=3 else
      "1000" when d=4 else
      "0001" when d=5 else
      "0011" when d=6 else
      "0101" when d=7 else
      "0111" when d=8 else
      "1001" when d=9 else
      "0000";
```

```
  -- ADD
  d3 <= ('0' & s) + ('0' & d2);

  -- REG2 och MOD10
  process(clk,rst)
  begin
    if rst='0' then
      s <= "0000";
    elsif rising_edge(clk) then
      if knapp='1' then
        if d3<10 then
          s <= d3(3 downto 0);
        else
          s <= d3(3 downto 0) + 6;
        end if;
      end if;
    end if;
  end process;

  -- K
  u <= d when siffra10='0' else
       "0000" when s=0 else
       10-s;

end Behavioral;
```

B.2 Testbänk för personnummermaskinen

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT add
        PORT(
                clk : IN std_logic;
                rst : IN std_logic;
                stb : IN std_logic;
                kb : IN std_logic_vector(3 downto 0);
```

```
                u : OUT std_logic_vector(3 downto 0)
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL rst :  std_logic;
        SIGNAL kb :  std_logic_vector(4 downto 0);
        SIGNAL u :  std_logic_vector(3 downto 0);

BEGIN

        uut: add PORT MAP(
                clk => clk,
                rst => rst,
                stb => kb(4),
                kb => kb(3 downto 0),
                u => u
        );


-- *** Test Bench - User Defined Section ***
   -- clk 1 Mhz
   tb : PROCESS
   BEGIN
      clk <= '0';
      wait for 0.5 us;
      clk <= '1';
      wait for 0.5 us;
   END PROCESS;

   -- reset
   rst <= '0', '1' after 0.4 us;

   -- tryck in pnr MSB=stb
   kb <=   "10101",               -- 5
           "00000" after 2.1 us,
           "10110" after 5.1 us,  -- 6
           "00000" after 7.1 us,
           "10000" after 10.1 us, -- 0
           "00000" after 12.1 us,
           "10110" after 15.1 us, -- 6
           "00000" after 17.1 us,
           "10000" after 20.1 us, -- 0
           "00000" after 22.1 us,
```

```
            "10110" after 25.1 us, -- 6
            "00000" after 27.1 us,
            "10001" after 30.1 us, -- 1
            "00000" after 32.1 us,
            "10010" after 35.1 us, -- 2
            "00000" after 37.1 us,
            "10011" after 40.1 us, -- 3
            "00000" after 42.1 us,
            "10000" after 45.1 us, -- 0
            "00000" after 47.1 us;

-- *** End Test Bench - User Defined Section ***

END;
```

Bilaga C

Projektkatalog för DK 2005

Här följer en del tips och förslag på möjliga projekt. Använd projekten som underlag och modifiera enligt egna önskemål. Kontrollera med handledare eller examinator om speciell utrustning behövs och om den i sådana fall finns tillgänglig.

Ett försök att gradera projekten efter svårighet har gjorts:

1. *stjärna* : Projektet bör gå att lösa med 2 CPLD-er.
2. *stjärnor* : Projektet bör gå att lösa med 4 CPLD-er.
3. *stjärnor* : Projektet bör gå att lösa med 4 CPLD-er och en del klurighet. Här bör nog någon i gruppen ha byggt en digital apparat förut.

I samtliga fall kommer även några andra kretsar att behövas, t ex minnen, klockgenerator och display.

Videospel

Konstruera ett enklare videospel, i form av ett actionspel eller något mer intelligenskrävande, för en eller flera spelare. Enklare spel i svartvitt rekommenderas. Vissa spel kan alternativt använda en oscilloskopskärm som spelplan.

Bilrally **

Ett spel för två personer. Två bilar startar jämsides och kontrolleras med fyra eller åtta knappar eller en styrspak: Vänster, höger, upp och ned samt eventuellt de fyra diagonalriktningarna däremellan. Styrkommandona anger accelerationen för bilen. I varje steg kan man bara ändra hastighetsvektorn ett steg i någon av de åtta riktningarna, vilket innebär att bilen har en viss tröghet. Bilderna av bilarna behöver inte vara särskilt realistiska, men man skall kunna se på skärmbilden åtminstone vems bilen är och gärna också åt vilket håll den kör. Rallybanan skall visas på skärmen. Vid krock med sargen kan bilen antingen diskvalificeras, eller tvingas stanna en stund, eller skickas tillbaka till startpositionen. Om bilarna krockar med

varandra kan båda stanna, eller hastighetsvektorn ändras slumpvis (sladd?). Målgång bör detekteras automatiskt. Ett lopp behöver inte köras över mer än ett varv.

Lerduveskytte **

Objekt av något slag (lerduvor, fåglar, harar, kor eller liknande) skickas i snabba banor över skärmen. Det gäller att träffa dessa med en hagelbössa, vilken lämpligen simuleras med en tryckknapp och en joystick, som styr ett på skärmen rörligt sikte. Skotten skall markeras med någon form av blaffa på skärmen, och gärna en ljudeffekt. Poängen kan räknas som träffprocent.

Andra videoprojekt

Konstruera en apparat som visar en bild på en videomonitor i något annat syfte än ett videospel.

TV-textning **

En rörlig bild från videokamera eller annan videokälla skall förses med textremsa nertill. Både en och två textrader skall kunna visas. Innehållet i textraderna kan läggas i EPROM, men bör kunna ändras åtminstone mellan två olika utseenden. Textremsan skall också kunna stängas av helt.

Life-maskin ***

Life är ett fascinerande matematiskt ”spel” som uppfanns på 1970-talet av den framstående matematikern David Conway. En uppsättning mycket enkla regler används för att simulera tillväxten i en idealiserad värld av *celler*. Cellerna kan enkelt modelleras som var sin pixel i en binär bild, där varje tänd pixel motsvarar en cell. Varje cell kan ha åtta grannar i en kvadratisk omgivning om 3x3 pixels. En cell överlever om den har två eller tre grannar. Om den har färre dör den av ensamhet, om den har fler dör den av trängsel. I en tom pixel som har exakt tre grannar föds en ny cell. För varje gång man utför dessa transformationer skapas en ny generation av celler. Vissa mönster av celler är stabila eller dör omedelbart, andra övergår i tusentals olika formationer innan de stabiliseras eller dör ut. Vissa speciella formationer är periodiska och övergår i ursprungsformen igen. En del formationer kan ”röra sig”, andra kan ”äta upp” sina grannar, och åter andra kan växa obegränsat.

Det har gjorts otaliga simuleringar av *Life* på mer eller mindre snabba datorer. Emellertid är reglerna så enkla att de utan svårighet kan implementeras direkt i hårdvara. Med ett snabbt och stort RAM-minne som spelplan och några parallellt arbetande skiftregister med tillhörande logik kan man generera life-generationer i binära bilder så snabbt som 10 miljoner pixels per sekund.

Konstruera en maskin som kan simulera *Life* och visa bilden på en videomonitor. Minst 256x256 celler skall simuleras. Hastigheten skall vara minst 25 generationer per sekund. Evolutionen skall kunna stannas tillfälligt och köras stegvis en

generation i taget. Något sätt att mata in ett startmönster måste också finnas, antingen från färdiga mindre mönster i ett EPROM eller från någon form av interface till ett tangentbord eller en persondator. Startmönstret kan också vara slumpmässigt, men i så fall kan man inte studera de mest intressanta formationerna.

Spel

Master Mind *

Spelet Master Mind anses vara bekant. Bygg en maskin som spelar den passiva rollen i spelet, det vill säga genererar en dold slumpmässig kombination och ger ledtrådar till spelarens gissningar. Färgerna får ersättas med siffror, bokstäver eller andra lämpliga symboler. Svårighetsgraden kan varieras med olika antal färger och olika många symboler i kombinationen.

Datorteknik

Nedanstående projekt kräver grundläggande kunskaper i datorteknik, ungefär motsvarande kursen Datorteknik på LiTH. De är dock inte nödvändigtvis mer komplicerade än andra projekt i katalogen.

Enkel dator **

Konstruera en enkel generell dator. Arkitekturen kan väljas fritt, men förslagsvis kan man välja en 8-bitars databuss, en 12 -eller 16-bitars adressbuss, två till åtta generella eller specialiserade register och en tämligen rudimentär instruktionsuppsättning med bara det allra nödvändigaste. Mikroprogrammering bör användas för att få rimligt enkel maskinkodsprogrammering. Data och program skall kunna lagras i RAM, men testprogrammen kan läggas helt i EPROM. En högt specialiserad datorarkitektur kan naturligtvis också byggas, exempelvis en stackdator eller en rent seriell dator eller projektet *Life-maskin* tidigare i projektkatalogen.

Enkel kalkylator **

Konstruera en enkel oktal, hexadecimal eller decimal kalkylator som åtminstone kan addera och subtrahera i en rymd av heltal med absolutbelopp upp till cirka 500. Overflow bör indikeras. Talen och operationerna bör kunna matas in från något slags tangentbord. Omvänd polsk notation med stackbaserade operationer kan användas, vilket gör uppgiften något enklare. Eventuellt kan en binär kalkylator byggas, med endast tangenterna 0 och 1 för inmatning av operanderna, men med minst åtta bitars operander och operationerna addition, subtraktion och/eller tvåkomplementsnegation, skift, AND, OR och NOT. Stacken kan gärna ha utrymme för fler än två operander, till exempel fyra eller åtta.

Enkel dator för Mastermind **

Denna uppgift löses egentligen bäst genom att att använda en enkel mikrokontroller. Din uppgift blir förstås att designa denna.

Enkel dator för Morsegenerator **

Enkel pipelinad dator ***

Konstruera en enkel pipelinad dator. Arkitekturen kan väljas fritt, men förslagsvis kan man välja en 16-bitars databuss, en 12 -eller 16-bitars adressbuss, två till fyra generella register. Program- och data-minne måste vara skilda åt. Mikroprogrammeringen förekommer inte i pipelinade datorer. Antalet pipelinesteg kan variera, men ett grundtips är tre.

Bilaga D

Att bygga en liten dator

D.1 En ackumulatormaskin

D.1.1 Allmänt

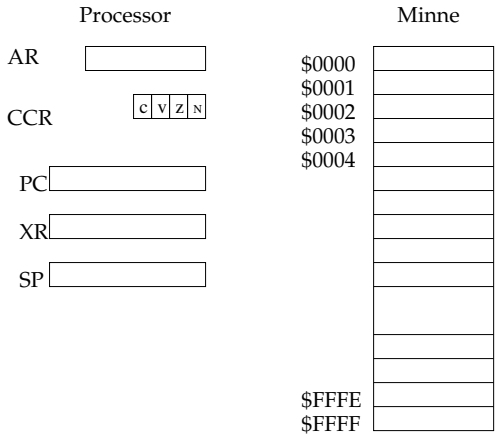
Detta avsnitt behandlar en fiktiv ackumulatormaskin, som mycket liknar Motorola M6800. Denna var en av de första mikroprocessorerna och dök upp 1975. Vi introducerar här begreppet arkitektur eller datorarkitektur. Med detta menar vi datorns *programmerarmodell* och *instruktionsrepertoar*. Ackumulatormaskinens arkitektur är den klart dominerande bland enklare mikroprocessorer, t ex PIC. Typiskt är ackumulatorregistret och minnet 8 bitar breda.

D.1.2 Programmerarmodell

Med programmerarmodell menar vi de register, som en maskinspråksprogrammerare använder. Lägg märke till att denna modell är en typisk abstraktion, lämplig att ta till vid programmeringen. Naturligtvis innehåller datorn mera hårdvara. Registren i fig. D.1 har följande funktion:

- *AR* = *ackumulatorregister* (8 bitar). Innehåller resultatet av den senaste beräkningen.
- *CCR* = *condition code register, villkorsregister*.
  - *C*. Carryflaggan = 1 omm föregående instruktion gett carry.
  - *V*. oVerflowflaggan = 1 omm föregående instruktion gett overflow.
  - *Z*. Zeroflaggan = 1 omm $AR == 0$.
  - *N*. Negativflaggan = 1 omm $AR < 0$, dvs $N = MSB(AR)$.

  Lägg märke till C,V ger info utöver innehållet i AR, till skillnad från Z,N.
- *PC* = *programräknare*. (16 bitar). Innehåller adressen till (pekar på) den instruktion, som står i tur att exekveras.



Figur D.1: Programmerarmodell för en enkel ackumulatormaskin.

- *XR = indexregister*. (16 bitar). Innehåller adressen till (pekar på) en tabell i minnet.
- *SP = stackpekare*. (16 bitar). Innehåller adressen till (pekar på) toppen av stacken, som bl a används vid subrutinanrop och avbrott.

Dessa register utgör den minimala uppsättningen för en användbar maskin. Vi lägger märke till att AR är ett dataregister medan PC,XR och SP är adressregister (pekarregister).

D.1.3 Instruktionsrepertoar.

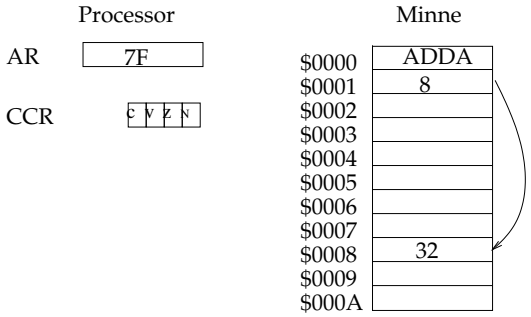
Ett maskinspråksprogram består av

- *instruktioner*. En instruktion består i sin tur av
  - operationskod (opkod), som förstås anger vad som skall göras.
  - operandutpekning, som anger var operanden finns. Denna utpekning kan vara mer eller mindre raffinerad. Man brukar tala om adresseringsmoder.
- *operander* (data).

Vi tar ett exempel

```
ADDA 8      ; AR = AR + M(8), dvs innehållet i minnescell 8
            ; adderas till ackumulatorn
```

Vi presenterar nu en instruktionslista:



Figur D.2: Instruktion ADDA 8 före exekveringen. Vad blir resultatet? Observera att alla flaggorna påverkas.

Mnemonic	Beskrivning	Funktion	Flaggor
`LDA A`	Load acc.	$AR = M(A)$	Z,N
`STA A`	Store acc.	$M(A) = AR$	
`ADDA A`	Add acc.	$AR = AR + M(A)$	C,V,Z,N
`SUBA A`	Subtract acc.	$AR = AR - M(A)$	C,V,Z,N
`INCA`	Increment acc.	$AR = AR + 1$	C,V,Z,N
`DECA`	Decrement acc.	$AR = AR - 1$	C,V,Z,N
`CMPA A`	Compare acc.	$AR - M(A)$	C,V,Z,N
`CLRA`	Clear acc.	$AR = 0$	Z=1
`ASRA`	Ar. shift right acc.	$AR = AR/2$	C,V,Z,N
`ASLA`	Ar. shift left acc.	$AR = AR \cdot 2$	C,V,Z,N
`LSRA`	Log. shift right acc.	$AR = AR >> 1$	C,V,Z,N
`ANDA A`	bitwise and acc.	$AR = AR\&M(A)$	Z,N
`ORA A`	bitwise or acc.	$AR = AR\|M(A)$	Z,N
`JMP A`	jump to A	$PC = A$	
`JMPC A`	jump to A if carry	$C = 1 \rightarrow PC = A$	
`JMPV A`	jump to A if overflow	$V = 1 \rightarrow PC = A$	
`JMPZ A`	jump to A if zero	$Z = 1 \rightarrow PC = A$	
`JMPN A`	jump to A if negative	$N = 1 \rightarrow PC = A$	

Vi är nu redo för ett litet assemblerprogram. Vi tänker oss att vi har en assemblator för vår maskin. Programmet ska beräkna medelvärdet av två tal med tecken. Dessa finns i minnescellerna 20 och 21. Vi antar för bekvämlighetens skull att alla siffror i vårt program är hexadecimala. Det korrekta medelvärdet ska placeras i cell 22. De flesta assemblatorer (även vår) vill ha källkoden snyggt inskriven i fyra fält: label, instruktion, operand och kommentar. Så här kan det se ut:

```
            ; mitt första asm-program
            ; kod
START:    LDA  20       ; hämta första operanden
          ADDA 21       ; lägg till den andra
```

```
          JMPV SPILL    ; spill?
          ASRA          ; dela med 2
          JMP HIT
SPILL:    ASRA          ; dela med 2
          ADDA #80      ; kompensera genom att flippa MSB
HIT:      STA 22
SLUT:     JMP SLUT      ; här stannar vi
           ; dataarea
          ORG 20
A:        DB 17
B:        DB 25
C:        DB 00
```

Några kommentarer till ovanstående program:

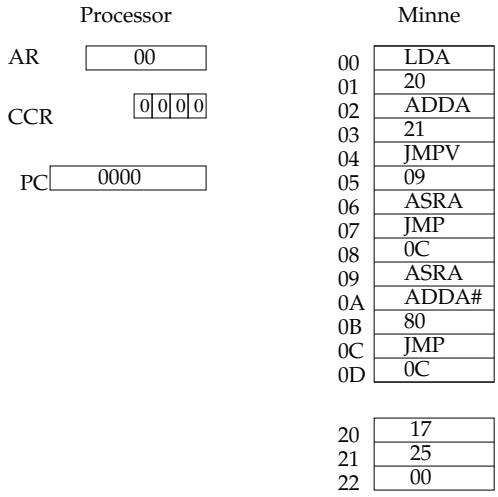
- Assemblatorn tillåter så kallade symboliska adresser. Innan programmet exekveras har dessa naturligtvis bytts ut mot de aktuella adresserna. Vanligtvis arbetar assemblatorn i två pass. I det första beräknas värdena på de symboliska adresserna, i det andra översätts alltihop till exekverbar kod.
- Vi introducerade *omedelbar* adresseringsmod. Observera den förödande skillnaden mellan `ADDA #80` ($AR = AR + 80$) och `ADDA 80` ($AR = AR + M(80)$). Vanligaste nybörjarfelet!
- `ORG` och `DB` är assemblerdirektiv, alltså ej instruktioner. `ORG 20` sätter lägespekaren (vid assembleringen) på 20. Nästa rad definierar en byte, 17, på denna adress. Om vi vill kan vi referera till byten med labeln A.
- Observera att instruktionen `ADDA 21`, påverkar spillflaggan. Vi testar värdet på `V` med `JMPV SPILL` och gör två olika saker beroende på utfallet. Observera att det inte går att testa på `V` efter `ASRA`. Då kan ju `V` ha fått ett nytt värde.

I fig. D.3 visas en programmerarmodell för spillexemplet. Vi tänker oss att vi har assemblerat programmet och placerat det på rätt ställe i minnet. I ett verkligt fall är naturligtvis även opkoderna utbytta mot hexkoder. Det är en mycket bra övning att singlesteppa assemblerprogram. Exekvera alltså instruktion för instruktion och skriv in ändrade värden i register och minnet.

D.1.4 Adresseringsmoder

Adresseringsmoder är alltså olika sätt att peka ut en operand. Man brukar använda begreppet effektiv adress EA, som helt enkelt är operandens minnesadress. Här kommer en lista på de moder som brukar förekomma:

1. *absolut*. Byten efter opkoden pekar ut operanden.
   Exempel: `LDA A ; AR = M(A)`. Effektiv adress $EA = A$.



Figur D.3: Programmerarmodell för spillexemplet.

2. *omedelbar*. Byten efter opkoden är operanden.
   Exempel: `LDA #A ; AR = A`. Effektiv adress $EA = PC + 1$ (här tänker vi oss att PC pekar på instruktionen).

3. *underförstådd*. Operanden ingår i opkoden. Exempel: `INCA ; AR = AR+1`. Här finns ingen effektiv adress.

4. *(minnes)indirekt*. Byten efter opkoden innehåller adressen till en pekare, som pekar på operanden.
   Exempel: `LDA (A) ; AR = M(M(A))`.
   Effektiv adress $EA = M(A)$. Denna mod är ovanlig, åtminstone för en enkel ackumulatormaskin. Den kan vara händig att ha vid hanterande av datastrukturer, t ex en array av pekare till strängar.

5. *indexerad*. Byten efter opkoden plus indexregistret bildar den effektiva adressen.
   Exempel: `LDA (XR)F ; AR = M(XR+F)`
   Detta är en mycket användbar mod, som asm-programmeraren inte kan leva utan!

6. *PC-relativ*. Byten efter opkoden plus programräknaren bildar den effektiva adressen. F är ett tal med tecken.
   Exempel: `BRA F ; PC = PC+F`
   BRA är alltså samma sak som JMP, men med PC-relativ istället för absolut mod. Lägg märke till att kod konsekvent skriven med PC-relativ mod blir lägesoberoende (relokerbar). Har vi absoluta hopp, går det ju inte att flytta på koden.

D.1.5 Indexerad adresseringsmod.

Vi tar en närmare titt på denna viktiga adresseringsmod. Vi behöver först några hjälpinstruktioner, som manipulerar XR.

Mnemonic	Beskrivning	Funktion	Flaggor
`LDX A`	Load XR	$XR = M(A), M(A+1)$	Z,N
`STX A`	Store XR	$M(A), M(A+1) = XR$	
`INX`	Increment XR	$XR = XR + 1$	C,V,Z,N
`DEX`	Decrement XR	$XR = XR - 1$	C,V,Z,N

Figur D.4: Instruktioner för hantering av indexregistret XR.

Vi illustrerar indexering med ett programexempel. I minnet, med början på adress 20, finns en tabell med 8 st tal utan tecken. Bilda summan av dessa. Vi antar nu att talen är så små att inte carry kan uppstå!

```
            ; min första indexering
START:      CLRA            ; summa=0
            LDX  #7         ; loopräknare=7
LOOP:       ADDA (XR)20     ; addera ett tal
            DEX             ; minska loopräknaren
            JMPP LOOP       ; tal kvar?
SLUT:       JMP SLUT
```

Några kommentarer till ovanstående program:

- Lägg märke till kommentarerna. De måste ge något som inte kan utläsas från opkoderna. Det är strängt förbjudet att kommentera den första raden med nn-ollställ ackumulatorn", det framgår ju av `CLRA`.
- Bakåthoppet måste gå till rätt rad. Om vi hoppar tillbaka till `LDX #7` fastnar vi i en oändlig loop.
- Vi gör alltså additionen för $XR = 7, 6, \cdots, 0$, det blir 8 gånger. När $XR = -1$, ska vi hoppa ur loopen, dvs fortsätta rakt nedåt. Här är det alltså viktigt att välja rätt hoppvillkor, alltså hoppa om positivt $N = 0$. Vi inser nu att vi behöver 4 nya hoppinstruktioner, som testar på om resp. flagga är noll. Alltså `JMPNC`, `JMPNV`, `JMPNZ` och `JMPP`.

D.2 Subrutiner.

Vi blev så nöjda med medelvärdesexemplet att vi skulle vilja använda det på fler ställen i ett större program. Vi vill nu göra om programmet till en generell *subrutin*. Vi tänker oss att de tre talen ligger efter varandra på godtyckligt ställe i minnet.

Adressen till det första talet är en *inparameter* till subrutinen. Så här kan det nu se ut:

```
START:    LDS #FARAWAY ; måste göras, annars ...
          ...
          LDX TP1      ; beräkna ett medelvärde
MV1:      JSR MEAN
          ...
          LDX TP2      ; beräkna ett annat medelvärde
MV2:      JSR MEAN
          ...
SLUT:     JMP SLUT
           ;
           ; min första subrutin
           ; IN: XR pekar på talen
MEAN :    LDA (XR)      ; hämta första operanden
          ADDA (XR)1    ; lägg till den andra
          JMPV SPILL    ; spill?
          ASRA          ; dela med 2
          JSR HIT
SPILL:    ASRA          ; dela med 2
          ADDA #80      ; kompensera genom att flippa MSB
HIT:      STA (XR)2     ; lagra medelvärdet
          RTS
```

Låt oss följa gången i programmet. Vi lägger adressen i XR och hoppar till subrutinen med `JSR`, jump to subroutine. Subrutinen gör vad den ska. I botten på subrutinen sker ett återhopp, `RTS`, return from subroutine, till instruktionen efter subrutinhoppet. Huvudprogrammet gör något med detta medelvärde (får vi väl anta). Sedan kommer ett nytt subrutinanrop. Lägg nu märke till att återhoppet inte går till samma ställe som det förra återhoppet. Innan vi ger oss på att förklara hur detta går till, konstaterar vi att programmet fungerar precis som om vi hade kopierat in subrutinen istället för de två `JSR`. Att låta assemblatorn kopiera in en rutin där den ska vara kallas för makro(assemblering). Att använda makron ger längre men snabbare kod.

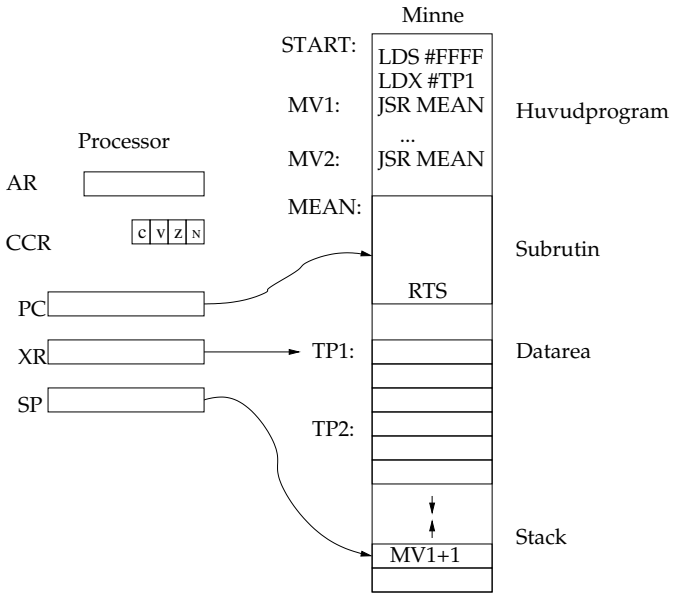
Vi behöver förstås några nya instruktioner:

Instruktionerna i fig. D.5 är alltså ganska komplexa. Återhoppsadressen måste sparas i minnet för att RTS ska hitta tillbaka. Detta ställe i minnet kallas för stacken och pekas ut av stackpekaren. Stacken är en datastruktur av typen LIFO = last in first out. Denna konstruktion gör att rekursion är möjlig, dvs en subrutin kan subrutinhoppa till sig själv, bara man får stopp på eländet. Observera noga vad `JSR` gör: 1) stackpekaren räknas ner 2) återhoppsadressen sparas på stacken och 3) hoppet sker. `RTS` gör alltihop i omvänd ordning: 1) återhopp till den adress som finns på toppen av stacken och 2) räkna upp stackpekaren.

Mnemonic	Beskrivning	Funktion	Flaggor
`JSR A`	jump to subroutine	$SP = SP - 1, M(SP) = PC, PC = A$	
`RTS`	return from subroutine	$PC = M(SP), SP = SP + 1$	
`LDS A`	load stackpointer	$SP = M(A), M(A + 1)$	
`PUSH`	push acc. to stack	$SP = SP - 1, M(SP) = AR$	
`PULL`	pull acc. from stack	$AR = M(SP), SP = SP + 1$	

Figur D.5: Instruktioner, som hanterar stacken. Vi använder här en predekrement stackpekare. SP pekar alltså på återhoppsadressen.

Programmet i fig. D.5 utnyttjar hela programmermodellen. I fig. D.6 visas situationen när `MEAN` exekverar första gången. Det är asm-programmerarens uppgift att hålla reda på allt som finns i minnet. Om man inte aktar sig kan stacken växa in i dataarean eller exekvering kan spåra ur och hamna i dataarean eller självmodifierande kod. På denna maskin finns det inget i arkitekturen som hindrar dessa otrevligheter.



Figur D.6: Programmerarmodellen när subrutinen `MEAN` exekverar första gången.

D.3 Mikroprogrammering.

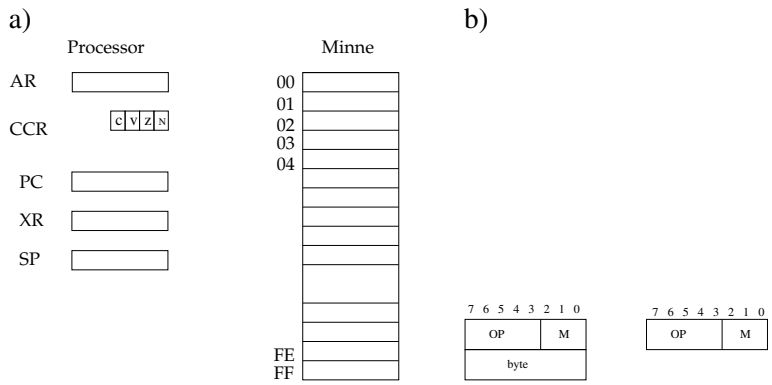
D.3.1 Allmänt

Mikroprogrammering är den lägsta nivå en dator kan programmeras på. Det är dock mycket ovanligt att vi på en kommersiell processor kommer åt mikroprogrammmet. Målsättningen med detta avsnitt är snarare att förstå hur en dator fungerar!

D.3.2 Begränsningar

För att datorn inte ska bli för svår att bygga inför vi en del begränsningar se fig. D.7:

- endast 8-bitars register. Detta är ju förstås ganska orealistiskt för pekarregistren eftersom det medför att vi endast kan ha $2^8 = 256$ minnesceller.
- endast två instruktionsformat. En instruktion består av
  - en byte för OPkod (5 bitar) och adresseringsmod (3 bitar). Detta räcker alltså till 32 opkoder och 8 adresseringsmoder. Inte helt orealistiskt!
  - eventuellt en extra byte, som innehåller ett tal, en adress eller en förskjutning.



Figur D.7: Begränsningar i den realiserade datorn. a) endast 8-bitars register i programmerarmodellen b) endast 2 instruktionsformat.

D.3.3 Adresseringsmoder och instruktionslista.

Vi kodar adresseringsmoderna på följande sätt:

M 3 bitar	Adresseringsmod	EA
0	absolut	A
1	indirekt	M(A)
2	index	XR+A
3	relativ	PC+2+d
4	omedelbar	PC+1
5	underförstådd	

Vi väljer att ta med följande instruktioner:

OPkod 5 bitar	Mnemonics	Verkan	Adresseringsmod	flaggor
0	LDA	AR=M(EA)	absolut, indirekt, index, omedelbar	ZN
1	STA	M(EA)=AR	absolut, indirekt, index	-
2	ADDA	AR=AR+M(EA)	absolut, indirekt, index, omedelbar	CVZN
3	SUBA	AR=AR-M(EA)	absolut, indirekt, index, omedelbar	CVZN
4	INCA	AR=AR+ 1	underförstådd	CVZN
5	DECA	AR=AR-1	underförstådd	CVZN
6	LDX	XR=M(EA)	absolut, indirekt, index, omedelbar	ZN
7	STX	M(EA)=XR	absolut, indirekt, index	-
8	INX	XR=XR+ 1	underförstådd	CVZN
9	DEX	XR=XR-1	underförstådd	CVZN
A	LDS	SP=M(EA)	absolut, indirekt, index, omedelbar	ZN
B	STS	M(EA)=SP	absolut, indirekt, index	-
C	INS	SP=SP+1	underförstådd	CVZN
D	DES	SP=SP-1	underförstådd	CVZN
E	PUSH	M(SP)=AR, SP=SP-1	underförstådd	-
F	PULL	SP=SP+ 1, AR=M(SP)	underförstådd	ZN
10	JMP	PC=PC+2+D	relativ	-
11	JMP	Om N=1 PC=PC+2+D annars PC=PC+2	relativ	-
12	JMPZ	Om Z=1 PC=PC+2+D annars PC=PC+2	relativ	-
13	JSR	M(SP)=PC+2,SP=SP-1 PC=PC+2+D	relativ	-
14	RTS	SP=SP+ 1, PC=M(SP)	underförstådd	-

Vi har här valt att särbehandla hoppen. Hoppen använder endast relativ mod. Inga andra instruktioner använder denna mod. Detta är ju förstås en fix som kommer att förenkla våra mikroprogramerarmödor.

Instruktionen `LDA 8` kodas:

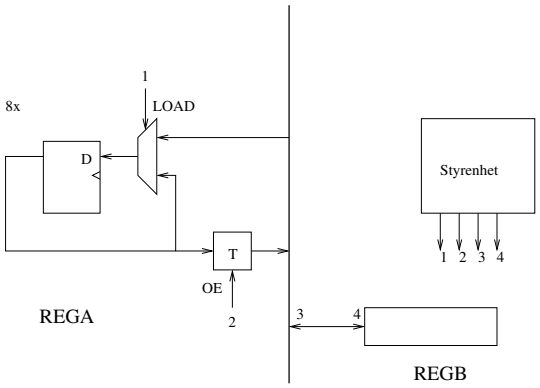
00000	000
0000 1000	

till skillnad från instruktionen `LDA #8` som kodas:

00000	100
0000 1000	

D.3.4 Komponenter

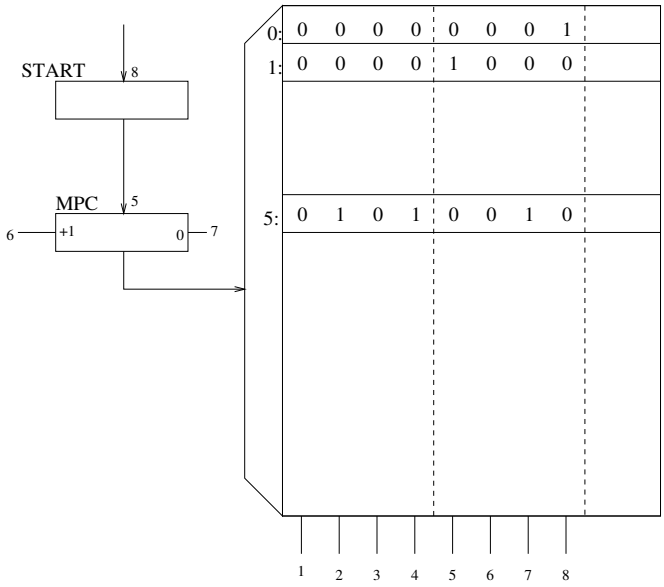
Vår mikroprogrammerade dator kan naturligtvis betraktas som ett sekvensnät, bestående av vippor och kombinatorik. Vi slår ihop vipporna i grupper om 8 till register, se fig. D.8. Registren har en `LOAD`-signal. Eftersom registren kopplas till en gemensam 8-bitars buss behöver vi också signalen `OE`, output enable. Vi bestämmer oss för att alla sådana verkställande signaler är aktivt höga. I fig. D.8 visas också ett



Figur D.8: Två 8-bitars register anslutna till en 8-bitars buss.

register `REGB` med förenklat skrivsätt, signal 3 är alltså `OE` och 4-an är `LOAD`. Om styrenheten aktiverar signalerna 2 och 4 under en klockperiod kopieras innehållet i `REGA` till `REGB`. Signalkombinationen 1 och 3 ger dataflödet motsatt riktning.

Nu är det alltså läge för att titta på en styrenhet. Det vi lärde oss i digitalteknikenen går utmärkt att använda här, speciellt konstruktion av SN med ROM. Det kommer att visa sig att vår styrenhet ska vara bra på att ge ifrån sig sekvenser av utsignaler, däremot behöver den inte reagera på insignalerna vid varje klockning. Vi föreslår alltså bygget i fig. D.9, uppbyggd av en universalräknare, som vi kallar *mikroprogramräknare* och ett ROM, kallat *mikrominne*. Varje rad i mikrominnet kallas en mikroinstruktion. `MPC` kan laddas, räkna upp eller nollställas beroende styrsignaler. Dessutom finns ett `START`-register, som håller en startadress för `MPC`. Signaler 1,2,3,4 går till registren i fig.D.8. Mikroprogrammet fungerar nu så här: rad 0 klockar in någonting i `START` och rad 1 laddar detta värde i `MPC`. Vi antar att vi nu hamnar på adressen 5 där följande sker: `REGB=REGA` och `MPC=0`. Mikroprogrammet börjar nu om på rad 0 och vi får väl tänka oss att `START` har fått ett nytt värde. Ovanstående är alltså ett exempel på enkel mikroprogrammering.



Figur D.9: En enkel styrenhet, bestående av *mikroprogramräknare* och *mikrominne*. Mikroinstruktionen består av två fält, registerstyrning och MPC-styrning.

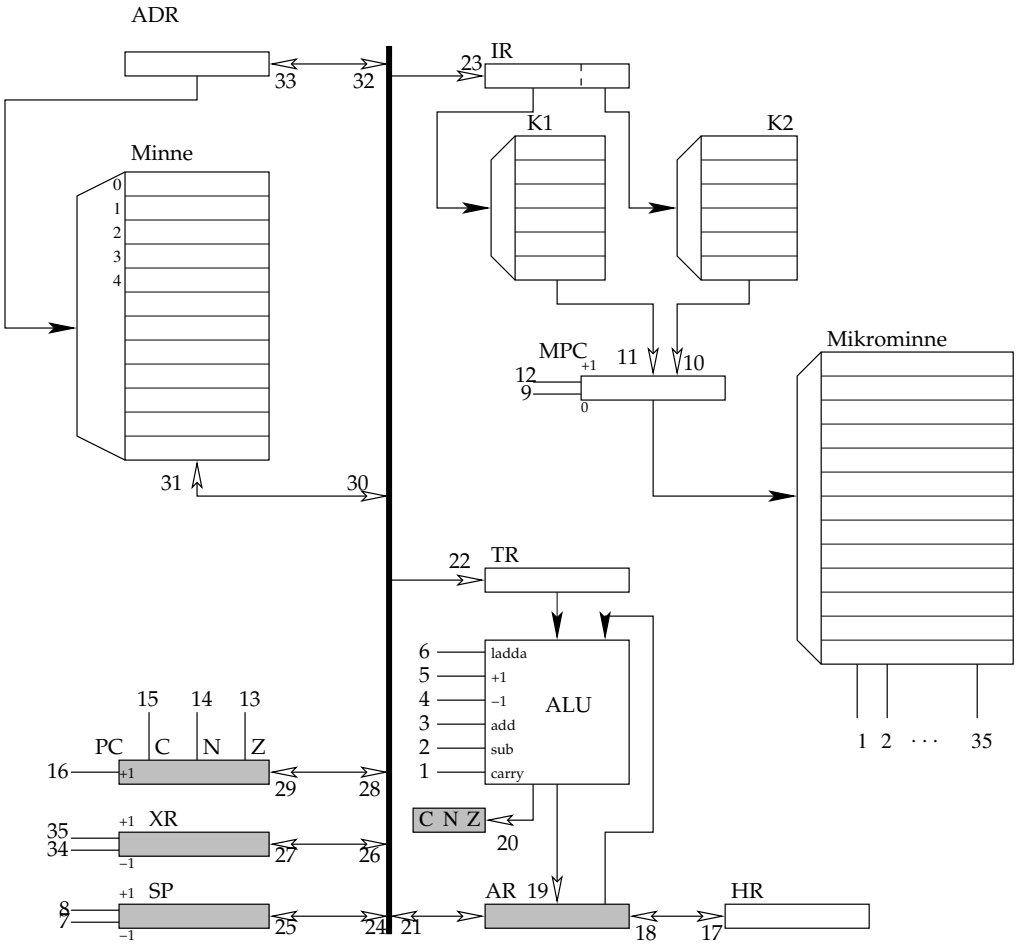
D.3.5 Mikrogrammerarmodell

Vi är nu redo att ge oss på mikroprogrammerarmodellen i fig. D.10. Vi anser att den består av tre större byggblock:

- *primärminne* med adressregistret `ADR`.
- *databehandlingsenhet* med de registret, som ingår i (asm)-programmerarmodellen. Dessutom behövs hjälpregistret `HR` och det temporära registret `TR`
- *styrenhet* med mikroprogramräknaren `MPC`, instruktionsregistret `IR`, mikrominne och två look-up tables `K1` och `K2`.

Innan vi ger oss på att skriva mikroprogram, konstaterar vi att vår dator arbetar i tretakt:

1. *hämtfas*. En instruktion hämtas till instruktionsregistret `IR`. Instruktionen pekas ut av `PC`. Innan vi lämnar denna är det alltså lämpligt att räkna upp `PC` och därefter hoppa till
2. *adressmodsfas*. Här kommer vi att behöva 6 olika mikroprogramsnuttar. Gemensamt för dessa är att operandens EA beräknas och placeras i `ADR`. Ett undantag finns dock, hoppadresser placeras i `TR`. Detta är möjligt eftersom hoppen (och endast hoppen) använder relativ adresseringsmod. Om vi har hämtat en byte från minnet räknar vi upp `PC` igen. Sedan sker hopp till



Figur D.10: Modell för mikroprogrammeraren. Komponenter, som ingick i asm-programmerarens modell är märkta med grått.

3. *exekveringsfas*. Här behövs lika många snuttar som instruktioner. Observera nu att dessa rutiner inte känner till vilken adresseringsmod, som använts i den föregående fasen. Undantaget är hoppen. Alltså, EA finns i `ADR`. Hämta operanden och gör något med den. Därefter hopp till rad 0 i mikroprogrammet.

Innehåll i mikrominnet

Vi visar här hämtfasen, 2 adresseringsmoder och 3 instruktioner.

Adress	Signaler	Funktion	Kommentar
0	28,33,16,12	PC->ADR, PC++, MPC++	Instruktionshämtning
1	30,23, 12	M->IR, MPC++	
2	10	K2 -> MPC	
3	28,33,16,12	PC->ADR, PC++, MPC++	Absolut adress
4	30,33,11	M->ADR, K1->MPC	
5	28,33,16,12	PC->ADR, PC++, MPC++	Indirekt adress
6	30,33,12	M->ADR, MPC++	
7	30,33,11	M->ADR, K1->MPC	
...			
20	30,22,12	M->TR, MPC++	LDA
21	6,20,19,9	TR->AR, CNZ, MPC=0	
22	21,31,9	AR->M, MPC=0	STA
23	30,22,12	M->TR, MPC++	ADDA
24	3,20,19,9	AR+TR->AR, CNZ, MPC=0	

Lägg alltså märke till hur mikroprogramsnuttarna är ihopkedjade:

- *instruktionshämtningen* avslutas med `K2->MPC`, dvs hopp till
- *adressmodsfasen*, som i sin tur avslutas med `K1->MPC`, dvs hopp till
- *exekveringsfasen*, som avslutas med `0 -> MPC`, dvs hopp till instruktionshämtningen för hämtning av nästa instruktion.

Innehåll i K1

K1 är en översättningstabell från OP-kod (5 bitar) till startadressen för respektive exekveringsfasen.

Adress	Innehåll	Kommentar
0	20	LDA
1	22	STA
2	23	ADDA
3	25	...

Innehåll i K2

K2 är en översättningstabell från koden för adresseringsmoden (3 bitar) till startadress för adresseringsmodsfasen.

Adress	Innehåll	Kommentar
0	3	Absolut adress
1	5	Indirekt adress
2	8	...

D.4 Implementering

Det är nu dags att implementera vår dator. Inför byggandet tänker vi så här

- Vi har gott om klockcykler. Det är inga som helst problem att vår mikroprogrammerade dator ska hinna med sina uppgifter. Vi betalar typiskt med klockcykler i stället för ytterligare en CPLD.
- Vi har ganska ont om plats i CPLD-erna. För varje VHDL-rad funderar vi på vad det kan tänkas bli för hårdvara.
- Vi vill inte vira så mycket. Det är ju enklare att ändra på felaktig kod än att vira om.

Vi är ganska nöjda med datorn i föregående kapitel, trots att den har några år på nacken. Vi ändrar lite i konstruktionen och inför ett I/O-block. En dator utan in/utmatning är ju ingen riktig dator. Vi tycker också att 8-bitars adressbuss är lite snålt och går upp till 12 bitar. Det betyder ju förstås att alla adressregister måste vara 12 bitar. Vi skär ner antalet adresseringsmoder till fyra, nämligen underförstådd, absolut, omedelbar och indexerad. Nu räcker det med två bitar för att koda adresseringsmoden.

Vi delar upp datorn i fem block (figur D.11):

1. *styrenheten*, som styr de fyra andra enheterna och sig själv. Vi har därför delat in raderna i mikrominnet i fem fält.
2. *ALU-enheten*, som dessutom innehåller ackumulatorregistret och in- och utregister.
3. *adressenheten*, bestående av programräknare, indexregister, stackpekare och adressregister.
4. *minnesenheten*, som egentligen utgörs av två minneskapslar, ett Flashminne och ett RAM.
5. *I/O-enheten*, som består av två parallelportar och en UART.

De fem kapslarna är ihopkopplade med en dubellriktad 8-bitars databuss `DBUS`.

D.4.1 Styrenheten

Vi hoppas att vi ska få in styrenheten i en CPLD 95108. Vi är dock lite oroliga för mikrominnet. Vi bestämmer oss nu för att minska antalet styrsignaler genom att använda ankodning. Istället för att ha individuella styrsignaler, har vi nu fält:

Fält styr	Verkan
000	Inkrementera MPC
001	Nollställ MPC
010	Ladda MPC från K1
011	Ladda MPC från K2
100	Ladda IR från DBUS

Här har vi alltså minskat antalet syrsignaler från 5 till 3. Lägg märke till att mikrominnet minskar men att vi måste kosta på avkodning på annat håll.

D.4.2 ALU-enheten

Vi fortsätter med ankodningstrategin.

Fält ALU	Verkan
0000	Ingenting
0001	AR - TR -> AR
0010	AR + TR -> AR
0011	AR - 1 -> AR
0100	AR + 1 -> AR
0101	TR -> AR
0110	HR -> AR
0111	uppdatera CNZ
1000	BUS -> TR
1001	AR -> DBUS
1010	AR -> HR
1011	...

Styrsignalen `1001` styr alltså tristatebufferten så att `DBUS <= AR`. I övriga fall måste vi förstås ha `DBUS <= "ZZZZZZZZ"`.

D.4.3 Adressenheten

Så här långt komna kan vi konstatera att en nackdel med ankodningsstrategin är att vi inte kan göra flera saker samtidigt. Varje rad i fältet gör ju bara en sak. För att åstadkomma det måste vi explicit "programmera" in det.

Fält areg	Verkan
0000	Ingenting
0001	DBUS -> ADR(7:0)
0010	DBUS(3:0) -> ADR(11:8)
0011	PC -> ADR
0100	XR -> ADR
0101	SP -> ADR
0110	DBUS -> PC(7:0)
0111	DBUS -> PC(7:0) om Z=1
...	...

Villkorsregistret `CNZ` måste dras till adressenheten för att detta ska fungera.

D.4.4 Minnesenheten

Här måste vi anpassa oss till kontrollsignalerna på de fysiska minneskapslarna. Ett ROM brukar ha Chip Enable (CE) och Output Enable (OE). Ett RAM har utöver dessa två signaler också Write Enable (WE). CE och OE kan kopplas ihop på respektive minne. Det borde alltså räcka med 3 styrsignaler till minnena. Vi har nu två minnen, program och data, med organisationen 256×8.

D.4.5 I/O-enheten

För att hantera I/O-enheten måste vi hitta på minst fyra nya instruktioner, nämligen:

- `IN`. Kopiera IN till AR.
- `UT`. Kopiera AR till UT.
- `RX`. Vänta på att ett tecken har mottagits och kopiera sedan UART till AR.
- `TX`. Vänta på att ett tecken har sänts och kopiera sedan AR till UART.

För att implementera UART-instruktionerna behövs också en smärre ombyggnad av styrenheten. Lämpligtvis införs villkorlig inkrementering av mikroprogramräknaren MPC. Vi går inte närmare in på detaljerna här, utan gissar att fem styrsignaler räcker för I/O-enheten. Vi landar i så fall på 18 styrsignaler, drygt hälften mot förut.

D.5 Simulering

Det är **jätteviktigt** att simulera hela datorn, allt i figur D.11, innan den byggs. Här ska vi därför visa hur det går till. Vi ska tillverka en testbänk, låt oss kalla den `helakonkarongen.vhd`, i vilken vi gör följande:

- instantierar *styrenhet*.
- instantierar *ALU*.
- instantierar *adressenhet*.
- instantierar *I/O-enhet*.
- instantierar simuleringsmodellen *ROM*, i vilken vi har lagt in ett trevligt program.
- drar igång klockan och gör en resetpuls.

```
entity helakonkarongen is
end helakonkarongen;

architecture behavior of helakonkarongen_vhd is
  -- Enheterna måste deklareras som component!
  component styrenhet
    port(
      clk,rst             : in    std_logic;
      dbus                : in    std_logic_vector(7 downto 0);
      styr_alu,styr_areg  : out   std_logic_vector(3 downto 0);
      styr_minne,styr_io  : out   std_logic_vector(4 downto 0);
```

```
    );
end component;
--
component alu
  port(
    clk,rst             : in    std_logic;
    dbus                : inout std_logic_vector(7 downto 0);
    styr_alu            : in    std_logic_vector(3 downto 0);
    inport              : in    std_logic_vector(7 downto 0);
    utport              : out   std_logic_vector(7 downto 0);
    cnz                 : out   std_logic_vector(2 downto 0);
    );
end component;
--
component areg
  port(
    clk,rst             : in    std_logic;
    dbus                : in    std_logic_vector(7 downto 0);
    abus                : out   std_logic_vector(11 downto 0);
    cnz                 : in    std_logic_vector(2 downto 0);
    );
end component;
--
component io
  port(
    clk,rst             : in    std_logic;
    dbus                : inout std_logic_vector(7 downto 0);
    styr                : in   std_logic_vector(4 downto 0);
    );
end component;
--
component rom
  port (
    a  : in  std_logic_vector (11 downto 0);
    d  : out std_logic_vector (7 downto 0);
    ce,oe : in  std_logic;
end component;

-- Interna signaler i testbänken
signal clk : std_logic := '0';
signal rst : std_logic := '1';
signal dbus, inport,utport : std_logic_vector(7 downto 0);
signal abus                : std_logic_vector(11 downto 0);
signal cnz : std_logic_vector(2 downto 0);
```

```
  signal styr_alu,styr_areg  :   std_logic_vector(3 downto 0);
  signal styr_minne  :   std_logic_vector(4 downto 0);

begin

  -- Styrenheten
  styrenhet0 : styrenhet port map(clk,rst,dbus,styr_alu,styr_areg,styr_mi

  -- ALU
  alu0 : alu port map(clk,rst,dbus,styr_alu, inport, utport, cnz);

  -- Adressenheten
  areg0 : areg port map(clk,rst,dbus,abus,cnz);

  -- I/O-enheten
  io0 : io port map(clk,rst,dbus,styr_io);

  -- Programminne
  rom0 : rom port map(abus,dbus,styrminne(0),styrminne(1));

  clk <= not clk after 0.5 us;
  rst <= '0' after 1 us;
  inport <= "10101010";
```

Så här kan simuleringsmodellen för ROM se ut.

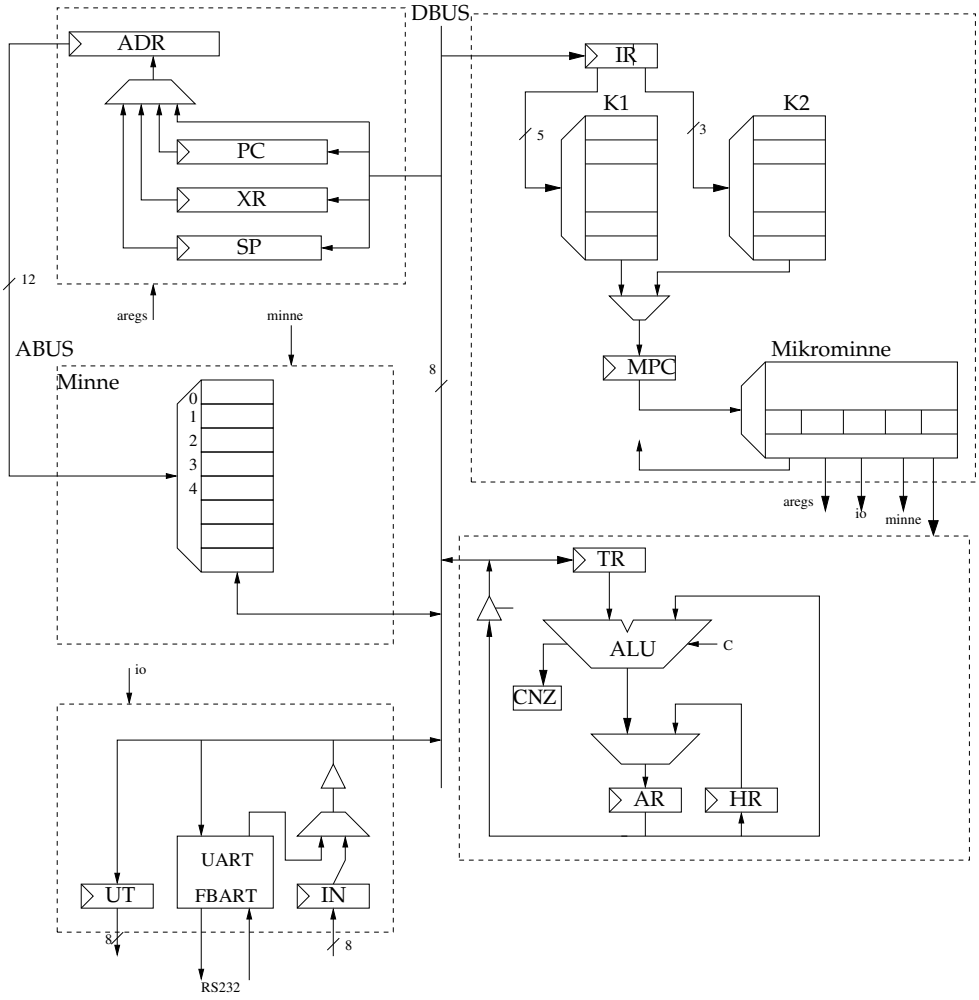
```
entity rom is
  port (a   : in  std_logic_vector (11 downto 0);
        d   : out std_logic_vector (7 downto 0);
        ce  : in  std_logic;
        oe  : in  std_logic);
end rom;

architecture Behavioral of rom is
  signal   dout : std_logic_vector (7 downto 0);
  type     memtype is array(0 to 1023) of std_logic_vector (7 downto 0);
  constant mem  : memtype :=
    (X"AB",                             -- 0: IN
     X"41",                             -- 1: INC
     X"B5",                             -- 2: OUT
     X"80",                             -- 3: JMP
     X"00",                             -- 4: 0
     others => X"00");
```

```
begin
  -- läsning
  d <= mem(CONV_INTEGER(a)) when ce = '0' and oe = '0' else "ZZZZZZZZ";
end Behavioral;
```

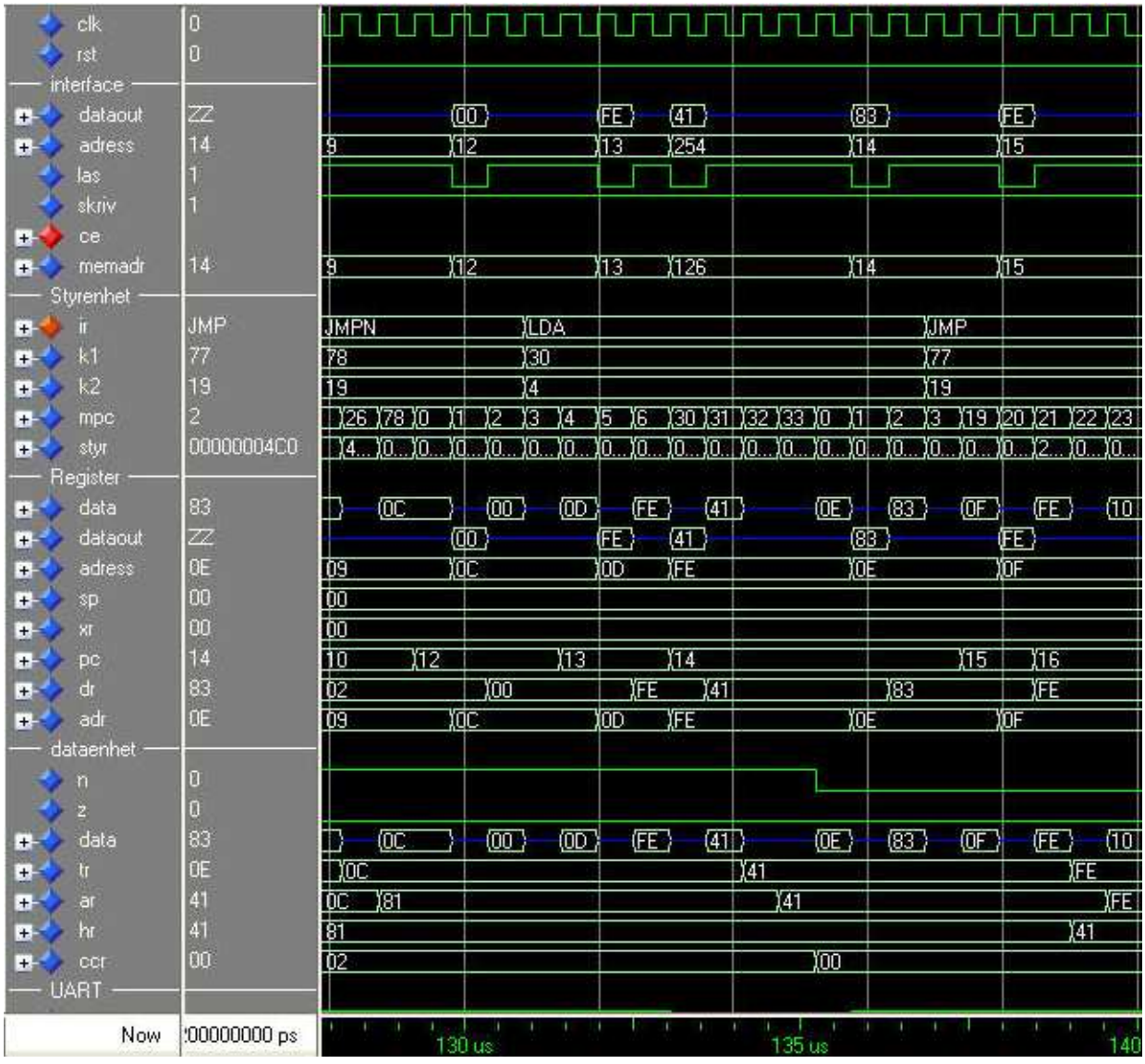
Nu är det bara att dra igång simuleringen och kolla att datorn exekverar kod, se figur D.12.

D.6 Några bilder

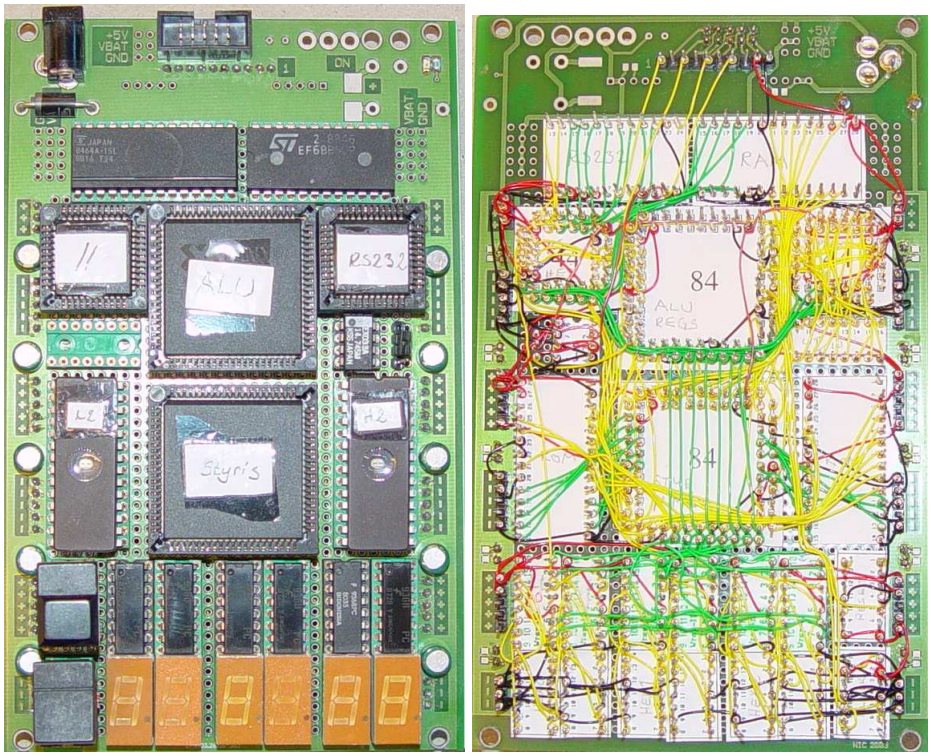
I figuren D.13 visas en färdigbyggd dator.



Figur D.11: Datorn uppdelad i fyra CPLD-er och en minnesenhet, som består av ett ROM och ett RAM. FBART är en UART, som du kan ladda ner från kursens hemsida.



Figur D.12: Ett exempel på simulering av en hel dator. Lägg märke till OP-koderna!



Figur D.13: Framsida och baksida av en liten dator. Lägg märke till den snygga virningen.

Litteraturförteckning

[1] Roos O. (1995): *Grundläggande datorteknik*, Studentlitteratur.

[2] Danielsson P-E och Bengtsson L. (1996): *Digital teknik*, Studentlitteratur.