Laborationer i Digitalteknik EITF65

2019

Namn: _____

Signeras av handledare vid godkännande

Lab 1: _____ Lab 4: _____

Lab 2: _____ Lab 5: _____

Lab 3: _____

LUNDS UNIVERSITET

Institutionen för Elektro- och Informationsteknik
Lund University
Printed in Sweden, 2019
E-huset, Lund

Innehåll

UTRUSTNING OCH GENOMFÖRANDE

Laborationerna är av konstruktionstyp, i hemuppgifterna skall ett antal maskiner konstrueras som du realiserar (bygger) under laborationen. Det är viktigt att vara väl förberedd inför varje laboration.

Kopplingsplattorna

Till varje laboration finns en kopplingsplatta. Kretsarna på plattorna finns listade först i varje laboration. Plattorna är uppdelade i moduler, en för varje krets, där de logiska symbolerna är utritade. Vid symbolernas ingångar och utgångar finns kopplingsstift som kan förbindas med sladdar.

Förutom moduler som innehåller rena standardkretsar, tex `74LS00` som innehåller 4 stycken NAND-grindar, finns på kopplingsplattorna ett antal specialmoduler. Dessa kan vara specifika för de olika laborationerna eller finnas på alla plattorna. På de flesta kopplingsplattorna finns moduler med lysdioder, omkopplare, klockenheter eller kopplingar till logikanalysatorn.

Lysdioderna används för att åskådliggöra signaler på kopplingsbordet. Till varje lysdiod finns ett kopplingsstift där signalen som skall visas kopplas in.

För att generera insignaler till konstruktionerna används vippströmbrytarna. Till varje omkopplare finns ett kopplingsstift. Om omkopplaren är uppfälld är det en logisk etta och om den nedfälld en logisk nolla.

Klockenheten innehåller dels en manuell klocka och dels en automatisk. Den manuella klockan består av en fjädrande omkopplare där utsignalen är kopplad så att inga kontaktstudsar uppstår. [1] Så länge omkopplaren hålls nere ges en logisk etta ut på kopplingsstiftet och då den släpps upp ges en logisk nolla. Den manuella klockan används framför allt för att testa systemet. När det skall köras kopplas den automatiska klockan in, antingen med 2 Hz eller 32,7 kHz.

[1] När en omkopplare sluts kommer kontakten att omväxlande sluta och öppna för att slutligen förbli sluten. Fenomenet kallas kontaktstudsar.

Kopplingsplattan ansluts till ett *spänningsaggregat* som distribuerar matningsspänning och jordpunkt till de olika kretsarna. Koppling får endast ske med spänningsaggregatet frånkopplat.

Laborationsförberedelser

Vid varje laborationstillfälle skall du ta med din laborationsbok och dina datablad. Bland annat krävs det att du bokför dina resultat i laborationsboken. Databladen behövs tex vid felsökning.

Varje laboration innehåller ett antal *hemuppgifter* invävda i texten. Dessa skall vara gjorda innan laborationen startar. Till samtliga realiseringar skall schema ritas.

Hemuppgift I.0.1

Detta är ett exempel på hur en hemuppgift markeras. Hemuppgifterna är alltid en del av förberedelserna. Allt som finns inom ramen skall vara gjort innan laborationen startar!

Godkännande

En laborationsassistent skall fylla i din laborationshandledning när du är klar. Detta görs genom att assistenten antecknar laborationsdatum och sin signatur på första sidan i manualen. *Kontrollera att det blir gjort!*

Varje laborationstillfälle varar fyra timmar. En ny uppgift markeras med följande linje.

Uppgift I.1. Start på uppgift

Markeringen, **I.1**, anger att det är uppgift **1** i laboration **I**. Samtliga uppgifter måste vara gjorda för att laborationen skall bli godkänd. Hinner du inte med uppgifterna på den tiden får du återkomma vid ett senare tillfälle. När uppgiften är färdig skall en handledare kontrollera kopplingen. Detta markeras som följer.

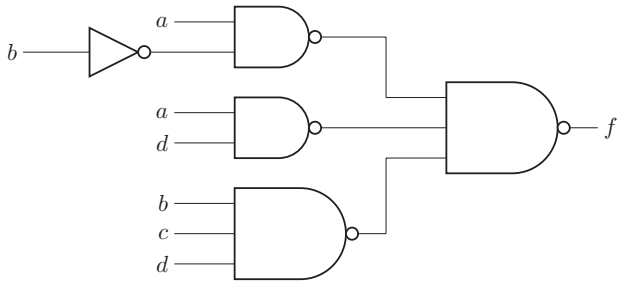
Slut på uppgift I.1

Råd vid realiseringen

Dina lösningar till uppgifterna resulterar ofta i att ett antal funktioner skall realiseras. Nedan följer ett förslag på de steg du bör göra då funktioner skall realiseras. På sidan två till varje laboration anges vilka kretsar som är tillgängliga, tillsammans med en bild på labplattan.

Steg 1: Kontrollera vilka kretsar som finns tillgängliga på laborationen. Tillgången på kretsar påverkar hur funktionen kan realiseras. Realiseringen skall vara sådan att de kretsar som är tillgängliga skall räcka för laborationens genomförande.

Steg 2: Rita nätet som realiserar funktionerna. Ha om möjligt insignalerna till vänster och utsignalerna till höger. Se figur I.1.



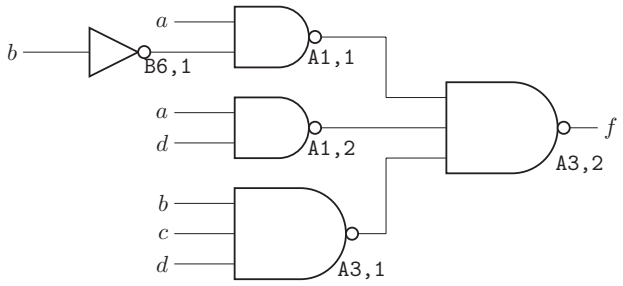
Figur I.1: Exempel på ett schema.

Steg 3: Markera i ditt schema var respektive grind finns på kopplingsplattan. Gör det tex genom att följa nedanstående steg:

Tänk dig att modulerna är placerade i ett koordinatsystem på kopplingsbordet. Översta raden kallar vi `A` och den understa för `C`. På varje rad numreras modulerna från vänster till höger: `1,2,...`. Numrera grindarna inom modulen, tex uppifrån och ner.

För varje grind i schemat anger du grindens placering på kopplingsplattan. Detta gör du genom att ange koordinaten för modulen som innehåller grinden, samt numret för grinden inom modulen.

För kopplingsplattan i laboration 1 blir numreringen som i figur I.2.



Figur I.2: Exempel på ett förbindelseschema.

Felsökning

Alla kommer att råka ut för felkopplingar under laborationerna. Då är det naturligt att felsöka i konstruktionen. Ett gott råd när du lokaliserar ett fel är att **leta bakifrån.**

För att illustrera vad som skall göras utgår vi från realiseringen av funktionen f i figur I.2. Antag att insignalerna, (a, b, c, d), har värdena $(0, 1, 1, 1)$. Då skall funktionen anta värdet 1 eftersom implikatorn bcd ger värdet 1.

Antag att realiseringen har ett fel som gör att utsignalen istället blir 0. Ett troligt fel är implikatorn bcd (kanske den rent av inte är inkopplad). Implikatorn realiseras i modulen `A3`. Vi finner inversen av implikatorn på utgången till `A3,1`. På det benet skall det finnas en 0:a. Signalen påverkar funktionen genom den nedersta ingången på `A3,2`. Observera att det mellan utgången `A3,1` och ingången `A3,2` finns en sladd som kan innehålla fel. För att hitta felet följer vi nedanstående exempel. Vi letar bakifrån, dvs från avvikelsen mot felkällan.

I realiseringen används NAND-grindar, dvs komplementet, $(bcd)'$, av implikatorn finns i realiseringen. För att f skall vara 1 måste ingång `A2:5` vara 0.

Om ingången på `A3,2` är 0 beror troligen felet på glappkontakt där kopplingssladden är ansluten eller att modulen `A3` är trasig. Båda dessa *kan* förekomma, men är ganska ovanliga.

Om ingången till `A3,2` är ett beror felet på någon tidigare bit i signalkedjan. Mät vid utgången från `A3,1`.

(i) Mäter du en **nolla på utgången från `A3,1`** har du troligen kabelbrott.

(ii) Mäter du en **etta på utgång `A3,1`** finns felet troligen vid realiseringen av implikatorn. Ställ dig följande frågor:

- Är alla variablerna anslutna?
- Har jag realiserat implikatorn korrekt?
- Fungerar modulen?

Ovanstående punkter ger dig för det mesta orsaken till ett fel. När felet är lokaliserat: **Åtgärda felet! Få fel är självreparerande.**

En naturlig fråga är nu: **Med vad mäter jag?** Varje laborationsplats är utrustad med en logikpenna, **TEST–it**. Denna har tre indikeringsdioder: **0**, **Puls** och **1**.

0: En grön lysdiod indikerar en logisk nolla. Om den uppmätta spänningen är under 0.8 volt är denna lysdiod tänd.

Puls: Om den uppmätta signalen växlar värde, oavsett frekvens, blinkar den gula lysdioden.

1: En röd lysdiod tänds om inspänningen är över 2.8 volt och indikerar en logisk etta.

Om den uppmätta signalen finns i intervallet 0.8 till 2.8 volt är ingen lysdiod tänd. Det är ett sätt att indikera en utgång som är i ett högimpedivt läge, exempelvis på en krets som har en så kallad *tristate*–utgång.

LABORATION 1

LEJONBUREN

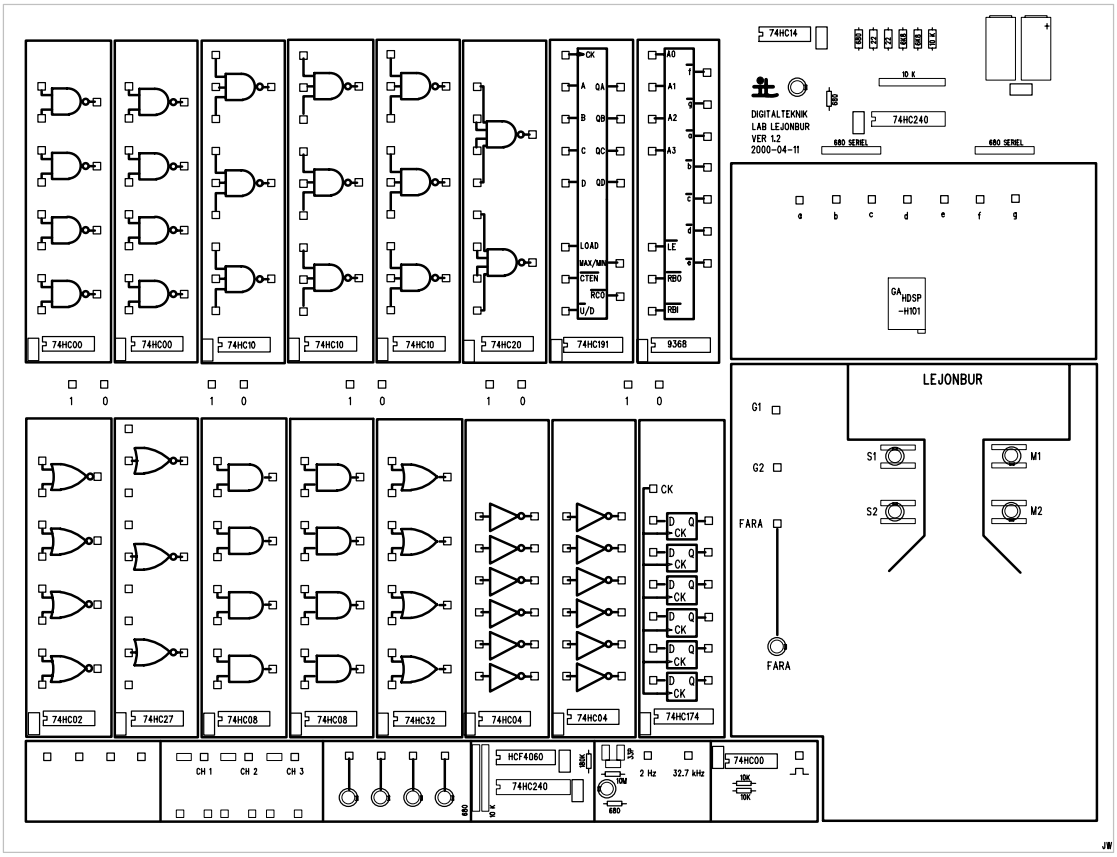
Laborationens syfte

Syftet med laborationen är dels att lära känna laborationsutrustningen och dels att få en uppfattning om hur en digital konstruktion är uppbygd. För att genomföra laborationen behövs *datablad*, dvs en specifikation över en färdig krets. De datablad som behövs för laborationerna finns i appendix A. Till att börja med skall några enkla funktioner realiseras med olika sorters grindar. Det ger en inblick i hur deMorgans lagar kan utnyttjas för att konvertera mellan olika sorters realiseringar. Därefter skall ett sekvensnät konstrueras. Här är det viktigt att förstå vad de olika delarna (kombinatoriskt nät och D-element) har för uppgifter i den färdiga realiseringen. Slutligen skall en standardkrets i form av en räknare användas för att utöka realiseringen.

Följande kretsar finns till laborationen:

Antal	Krets	Beskrivning
2	7400	4 st 2 ingångars NAND
3	7410	3 st 3 ingångars NAND
1	7420	2 st 4 ingångars NAND
2	7408	4 st 2 ingångars AND
1	7402	4 st 2 ingångars NOR
1	7427	3 st 3 ingångars NOR
1	7432	4 st 2 ingångars OR
2	7404	6 st inverterare
1	74174	6 st D-element
1	74191	U/D-räknare
1	9368	Drivkrets för display
1	HDSP-H101	Display
1		Lejonbur

Kretsarnas placering på labkortet



I den första uppgiften skall en funktion realiseras med kombinatoriska nät baserade på olika typer av grindar. För att se hur det kan utföras behövs först deMorgans lagar. Dessa skall visas med hjälp av funktionstabeller som sedan verifieras under laborationen.

Hemuppgift 1.0.1

Skriv ditt namn i rutan på första sidan i labbhäftet.

Hemuppgift 1.0.2

Läs labhäftets första kapitel om *Utrustning och genomförande*. Det är fyra sidor som är nyttiga för alla laborationerna.

Uppgift 1.1. Booleska funktioner

Hemuppgift 1.1.1

Visa med hjälp av funktionstabellen nedan att uttrycken

$$a' \wedge b' \quad \text{och} \quad (a \vee b)'$$

realiserar samma funktion.

a	b	a'	b'	$a' \wedge b'$	$(a \vee b)'$
0	0				
0	1				
1	0				
1	1				

Koppla upp och verifiera själv föregående hemuppgift.

Hemuppgift 1.1.2

Visa med hjälp av funktionstabellen nedan att uttrycken

$$a' \vee b' \quad \text{och} \quad (a \wedge b)'$$

realiserar samma funktion.

a	b	a'	b'	$a' \vee b'$	$(a \wedge b)'$
0	0				
0	1				
1	0				
1	1				

Koppla upp och verifiera själv föregående hemuppgift.

De framtagna ekvationerna

$$a' \wedge b' = (a \vee b)'$$
$$a' \vee b' = (a \wedge b)'$$

kallas deMorgans lagar. Med hjälp av induktion kan en mer generell variant visas:

$$\bigwedge_{i=1}^{n} x_i' = \left(\bigvee_{i=1}^{n} x_i \right)'$$
$$\bigvee_{i=1}^{n} x_i' = \left(\bigwedge_{i=1}^{n} x_i \right)'$$

Anmärkning: För att underlätta uttrycken används ofta tecknen $\bigvee$ och $\bigwedge$ på samma sätt som tex $\sum$ och $\prod$. Dvs

$$\bigvee_{i=1}^{n} x_i = x_1 \vee x_2 \vee \ldots \vee x_n$$
$$\bigwedge_{i=1}^{n} x_i = x_1 \wedge x_2 \wedge \ldots \wedge x_n = x_1 x_2 \ldots x_n$$

Som ovan utelämnas ofta symbolen för AND, $\wedge$, för att uttrycken skall bli mer lättöverskådliga.

En viktig följd av deMorgans lagar är följande likheter[2]

$$\bigvee_i \Big(\bigwedge_k x_k \Big) = \Big[\bigwedge_i \Big(\bigwedge_k x_k \Big)' \Big]'$$
$$\bigwedge_i \Big(\bigvee_k x_k \Big) = \Big[\bigvee_i \Big(\bigvee_k x_k \Big)' \Big]'$$

Det innebär att en funktion angiven som en OR-summa av AND-produkter kan realiseras som en NAND-produkt av NAND-produkter och att en funktion angiven som en AND-produkt av OR-summor kan realiseras som en NOR-summa av NOR-summor. Detta kommer att användas på senare laborationer eftersom det i stort sett bara finns NAND-grindar för realisering av kombinatoriska nät.

Det finns flera viktiga räkneregler för algebraisk förenkling av booleska uttryck. Exempelvis kan följande regler vara nyttiga (de kan enkelt visas med sanningstabeller).

$$ab = ba \qquad a \vee b = b \vee a$$
$$a(b \vee c) = ab \vee ac \qquad a \vee (bc) = (a \vee b)(a \vee c)$$
$$a \vee ab = a \qquad a(a \vee b) = a$$

Hemuppgift 1.1.3

Visa (algebraiskt) med hjälp av räknereglerna ovan att följande funktionsuttryck är lika med varandra.

$$f_1 = a'c' \vee c'd \vee ab'd$$
$$f_2 = \Big((a'c')'(c'd)'(ab'd)' \Big)'$$
$$f_3 = (a \vee c')(b' \vee c')(a' \vee d)$$
$$f_4 = \Big((a \vee c')' \vee (b' \vee c')' \vee (a' \vee d)' \Big)'$$

[2]För att komma fram till dessa likheter behöver man använda att två inverteringar tar ut varandra, dvs att $f = (f')'$.

Hemuppgift 1.1.4

Realisera (rita) näten i föregående hemuppgift.

Koppla upp och verifiera näten för de booleska uttrycken f_1 och f_2 i hemuppgift 1.1.3. Ta EJ bort det första nätet när detta kopplats upp och verifierats.

Kalla på en laborationshandledare som skall godkänna uppgiften.

Slut på uppgift 1.1

Uppgift 1.2. Lejonburen

I en djurpark finns en avdelning för lejon. Den består av två delar, dels en bur där lejonen kan gå in och dels en inhägnad som är en stor öppen plats där lejonen kan ströva runt. Inhägnaden är kuperad vilket gör den svåröverskådlig. Eftersom djurskötaren vill kunna städa inhägnaden medan lejonen är i buren behöves ett system där en lampa, `Fara`, lyser om något lejon är ute ur buren.

Vid passagen mellan buren och inhägnaden finns två givare, $\mathtt{G}_1$ och $\mathtt{G}_2$. Dessa är realiserade med fotoceller som är placerade med någon decimeters mellanrum och på lagom höjd. Se figur 1.1. Lejonen är ca två meter långa och kan inte passera gången samtidigt. De kan inte heller vända eller stanna i gången.

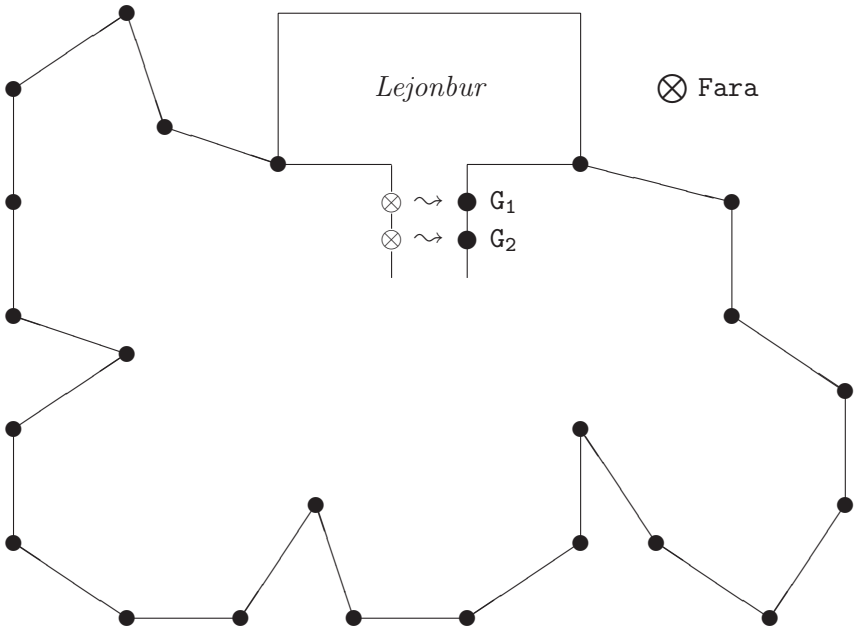
Då en ljusstråle bryts ger givaren en logisk etta medan den ger en logisk nolla om den inte bryts. Det innebär att när ett lejon går från buren till inhägnaden ger givarna följande sekvens:

$$(\mathtt{G}_1, \mathtt{G}_2) : 00 \to 10 \to 11 \to 01 \to 00$$

På samma sätt blir sekvensen när ett lejon går från inhägnaden till buren:

$$(\mathtt{G}_1, \mathtt{G}_2) : 00 \to 01 \to 11 \to 10 \to 00$$

Sekvenserna ovan markerar endast omslag av givarna. I realiseringen måste hänsyn tas till att det kommer ett antal upprepningar av varje kombination.



Figur 1.1: Skiss över lejonhägnet i en djurpark.

För att förenkla uppgiften skall först systemet konstrueras för ett lejon. Senare kommer det att utvidgas så att djurparken kan ha flera lejon.

Hemuppgift 1.2.1

Rita en graf för ett sekvensnät som kan hålla reda på *ett* lejon. Insignalerna skall vara signalerna från givarna $\texttt{G}_1$ och $\texttt{G}_2$ och utsignalen skall vara `Fara`. Lampan skall vara tänd (logisk etta) då lejonet är i gången eller ute i inhägnaden. Hur många tillstånd behövs?

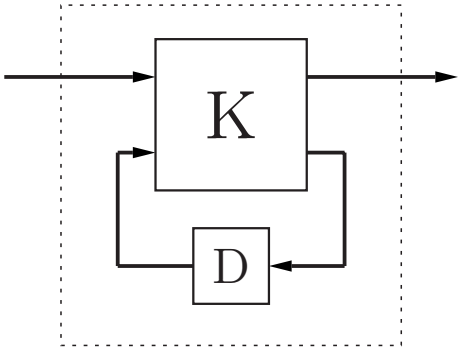
Eftersom tillstånden i en graf är en (abstrakt) beskrivning av det förflutna måste en realisering ha ett minne. Detta löses med hjälp av D-vippor[3]. En D-vippa har en dataingång, en klockingång och en utgång. Det värde som ligger på dataingången då klocksignalen går från logisk nolla till logisk etta kommer att finnas på utgången till nästa gång klocksignalen slår om från nolla till etta. I ett sekvensnät skall alla D-vippor alltid klockas med samma klocka! Klocksignalen är det som styr tillståndsuppdatering. Då klocksignalen slår om från nolla till etta hoppar processen ett steg i grafen.

För att representera tillstånden i D-vipporna behövs ett antal binära tillståndsvariabler, ofta betecknade med $q_i,\ i = 0, 1, \ldots, N-1$ där N är antalet tillståndsvariabler. Antalet tillståndsvariabler måste uppfylla

$$N \geq \lceil \log_2(\#\text{tillstånd}) \rceil.$$

I figur 1.2 är en schematisk bild för en sekvensmaskin uppritad. Den består av två delar; ett kombinatoriskt nät (K) och ett minne (D). Minnet består av N stycken D-vippor som vardera håller en tillståndsvariabel. Insignaler till det kombinatoriska nätet är dels insignalerna till sekvensnätet och dels tillståndsvariablerna. På samma sätt är dess utsignaler från det kombinatoriska nätet utsignalerna från sekvensnätet och nästa tillståndsfunktionerna (ingångarna på D-vipporna).

[3] D-vippan är den krets som realiserar D-elementen. De vippor som används i denna laborationen reagerar då klocksignalen slår om från nolla till etta. Det kallas att de är positivt flanktriggade och betecknas ofta med symbolen $\uparrow$.



Figur 1.2: Modell för realisering av sekvensmaskin.

Hemuppgift 1.2.2

Specificera en funktionstabell för ett kombinatoriskt nät som skall användas för att realisera grafen i hemuppgift 1.2.1. Skriv upp booleska uttryck för funktionerna.

Hemuppgift 1.2.3

Realisera (rita) sekvensnätet i föregående hemuppgift.

Koppla upp ett kombinatoriskt nät som realiserar de booleska uttrycken i hemuppgift 1.2.2. Koppla insignalerna från vippströmbrytarna och utsignalerna till lysdioder.

Koppla om så att insignalerna till sekvensnätet kommer från givarna $\texttt{G}_1$ och $\texttt{G}_2$. Kontrollera att det kombinatoriska nätet beter sig som det är specificerat i grafen.

Lägg till D-vipporna till kopplingen. Klocka kretsen för hand med den manuella klockan.

När sker uppdatering av tillstånden q_i?

Svar: _____

När sker uppdatering av utsignalen `Fara`? Måste kretsen klockas för att utsignalen ska ändras?

Svar: _____

När sker uppdatering av nästa tillstånds-funktionerna q_i^+?

Svar: _____

Byt ut den manuella klockan mot en automatisk (32.7kHz) och prova sekvensnätet.

Kalla på en laborationshandledare och visa kopplingen.

Slut på uppgift 1.2

Uppgift 1.3. Många lejon

För att hålla reda på mer än ett lejon skall en *räknare* användas. På labkortet finns standardkretsen 74191. Det är en modulo 16 räknare som kan räkna ett steg upp eller ner ($+1$ eller -1). Kretsen har två insignaler, $\overline{\texttt{CTEN}}$ och $\overline{\texttt{U}}/\texttt{D}$, som används som styrsignaler. När signalen $\overline{\texttt{CTEN}}$ är ett står räknaren stilla och då $\overline{\texttt{CTEN}}$ är noll räknar den ett steg. För att bestämma vilket håll räknaren skall räkna används signalen $\overline{\texttt{U}}/\texttt{D}$. Då $\overline{\texttt{U}}/\texttt{D}$ är noll räknar den upp ($+1$) och då $\overline{\texttt{U}}/\texttt{D}$ är ett räknar den ner (-1), se tabell 1.1.

Förutom ingångarna $\overline{\texttt{CTEN}}$ och $\overline{\texttt{U}}/\texttt{D}$ finns $\overline{\texttt{LOAD}}$, A, B, C, D och `CLOCK`. Signalen `CLOCK` bestämmer, som tidigare, när räknaren skall räkna. Det enda som inte styrs av klocksignalen är $\overline{\texttt{LOAD}}$, som är asynkron (inte beroende av klockan). Så fort $\overline{\texttt{LOAD}}$ får värdet noll kommer värdet som finns på insignalerna A till D att laddas till räknarens värde, eller tillstånd. Värdet som specificeras av insignalerna A till D är

$$V_{A-D} = A + B \cdot 2 + C \cdot 2^2 + D \cdot 2^3$$

På samma sätt anger utsignalerna Q_A till Q_D räknarens värde, eller tillstånd.

I tabell 1.1 är beteendet för räknaren specificerat. Härifrån, och från databladet, kan man utläsa att om $\overline{\texttt{CTEN}} = 0$ uppdateras räknaren då `CLOCK` går från noll till ett.

För att enkelt få ut räknarens värde finns en drivkrets för en display på kopplingsbordet, `9368`. Koppla från räknaren till drivkretsen `QA`→`A0`, `QB`→`A1`, `QC`→`A2` och `QD`→`A3`, se figur 1.3. Från drivkretsen till displayen kopplas `a`→`a`, ..., `g`→`g`. Displayen visar nu siffror ur det hexadecimala talsystemet, dvs 0, 1, 2, ..., 9, A, B, C, D, E, F.

Värde	$\overline{\texttt{LOAD}}$	A-D	CLOCK	$\overline{\texttt{CTEN}}$	$\overline{\texttt{U}}/\texttt{D}$	Nytt värde
–	0	A-D	–	–	–	A-D
$\boldsymbol{Q}$	1	–	↑	1	–	$\boldsymbol{Q}$
$\boldsymbol{Q}$	1	–	↑	0	0	$\boldsymbol{Q}+1 \mod 16$
$\boldsymbol{Q}$	1	–	↑	0	1	$\boldsymbol{Q}-1 \mod 16$

Tabell 1.1: Specifikation för standardkretsen 74LS191. Markeringen ↑ betyder övergången från noll till ett och markeringen – att värdet inte spelar någon roll (don't care).

Hemuppgift 1.3.1

Betrakta räknarens värde som dess tillstånd. Rita en graf för räknaren där dess tillståndsövergångar och insignalerna $\overline{\texttt{U}}/\texttt{D}$ och $\overline{\texttt{CTEN}}$ är markerade. Utsignaler behöver inte anges eftersom dessa är samma som tillstånden. Eftersom $\overline{\texttt{LOAD}}$ inte reagerar på klockan ska den inte inkluderas i grafen.

Till kopplingen behövs slutligen ett *styrnät* som kan ses som gränssnittet mellan givarna ($\texttt{G}_1$, $\texttt{G}_2$) och själva räknaren, se figur 1.3. Det är detta styrnät som skall konstrueras. Det skall känna skillnad på sekvenserna

$$(\texttt{G}_1, \texttt{G}_2)_{\text{ut}} = 00 \to 10 \to 11 \to 01 \to 00$$

och

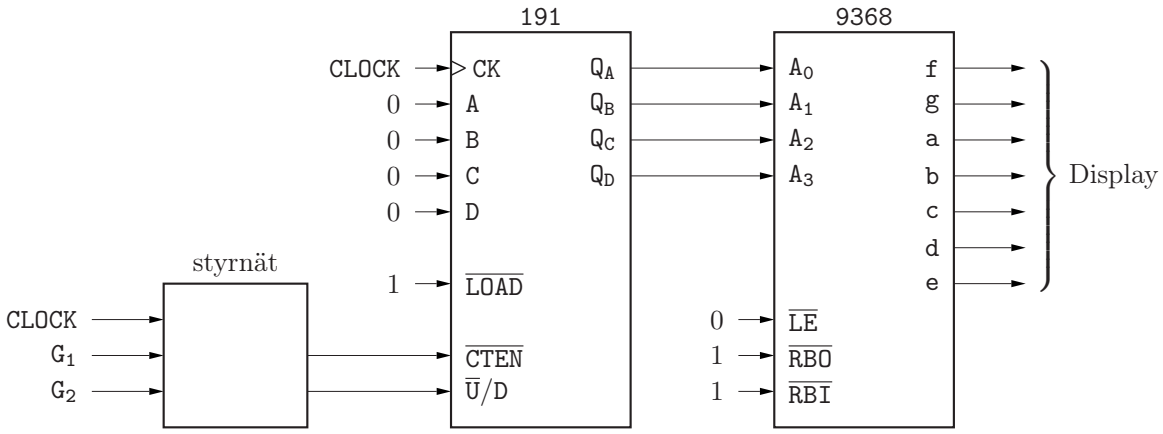
$$(\texttt{G}_1, \texttt{G}_2)_{\text{in}} = 00 \to 01 \to 11 \to 10 \to 00$$

Om $(\texttt{G}_1, \texttt{G}_2)_{\text{ut}}$ uppträder skall räknaren räkna upp ett steg och om $(\texttt{G}_1, \texttt{G}_2)_{\text{in}}$ uppträder skall den räkna ner ett steg. Motsvarande utsekvenser är

$$(\overline{\texttt{CTEN}}, \overline{\texttt{U}}/\texttt{D})_{\text{ut}} = \ldots 10 \to 00 \to 10 \ \ldots$$

och

$$(\overline{\texttt{CTEN}}, \overline{\texttt{U}}/\texttt{D})_{\text{in}} = \ldots 10 \to 01 \to 10 \ \ldots$$



Figur 1.3: Beskrivning hur styrnätet och räknaren skall kopplas ihop.

Som tidigare markerar givarvärdena $(G_1, G_2)_{\text{ut}}$ och $(G_1, G_2)_{\text{in}}$ upprepningar av värdena i realiseringen. Däremot får utsekvenserna bara ge ett räknevillkor till räknaren.

Hemuppgift 1.3.2

Rita en graf som talar om när något lejon går in eller ut ur buren (lejonet kan inte vända eller stanna i gången). Använd som tidigare givarnas signaler G_1 och G_2 som insignaler. Grafens utsignaler skall gå direkt till räknaren, använd därför $\overline{CTEN}$ och $\overline{U}/D$ direkt som de är beskrivna ovan. (Det går att lösa uppgiften med två tillstånd.)

Hemuppgift 1.3.3

Ange i en funktionstabell hur utsignalerna (signaler till räknaren och nästa tillståndsfunktion) beror på insignalerna (givarna och tillståndsvariablerna). Skriv upp booleska uttryck för utsignalfunktionerna och nästa tillstånds-funktionen samt realisera (rita) sekvensnätet.

Koppla upp det kombinatoriska nätet. Koppla insignalerna från vippströmbrytarna och utsignalerna till lysdioder. Kontrollera att det stämmer med tabellen i hemuppgiften.

Koppla in *D*-vippor och givare till kopplingen. Klocka med den manuella klockan för att kontrollera att sekvensnätet fungerar.

Koppla därefter ihop styrnätet med räknaren. Använd fortfarande den manuella klockan och kontrollera att konstruktionen fungerar.

Byt den manuella klockan mot 32.7 kHz klockan och kontrollera.

Hur många lejon kan nätet hålla reda på?

Svar: _____

Fortfarande vill djurskötaren ha en lampa `Fara` som lyser då minst ett lejon är ute i inhägnaden. Hur kan funktionen `Fara` realiseras?

Svar: _____

Utöka kopplingen med funktionen `Fara`.

Kalla på en laborationshandledare för att få uppgiften godkänd.

Slut på uppgift 1.3

LABORATION 2

ARITMETISK OCH LOGISK ENHET

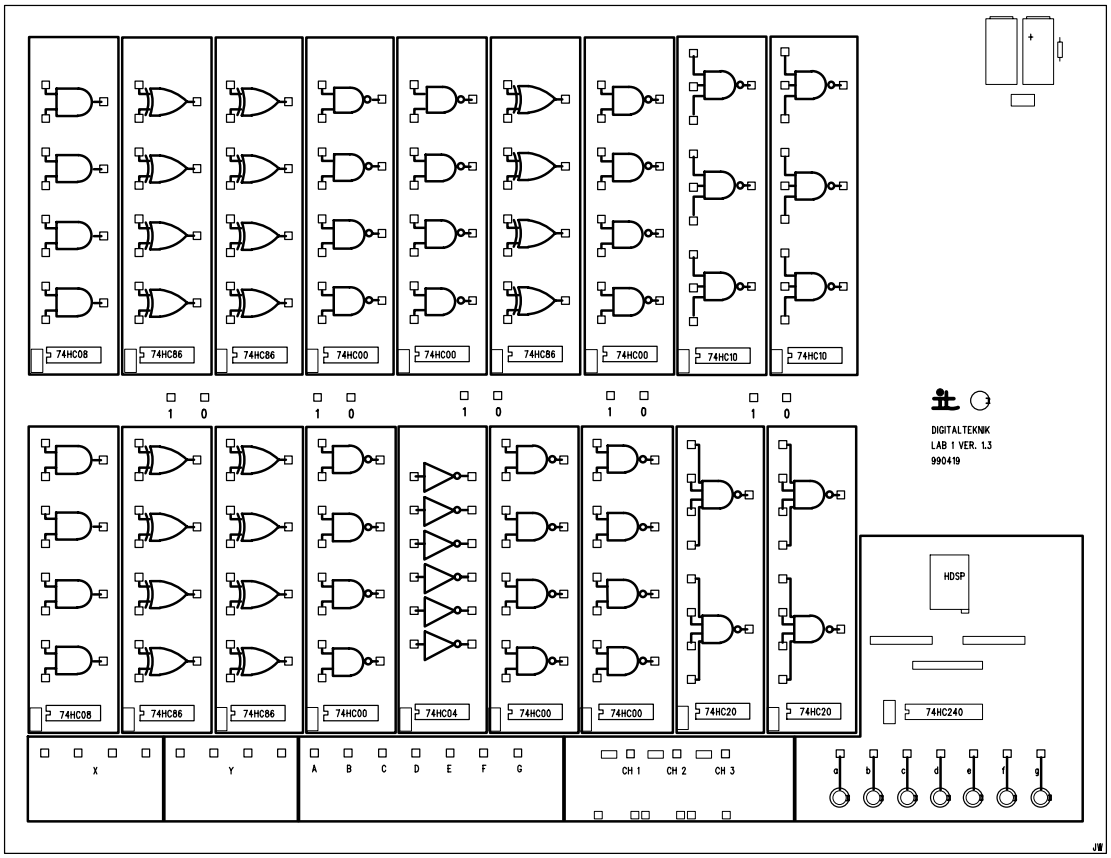
Laborationens syfte

Du skall få en viss känsla för funktion och storlek av kombinatoriska nät. Under laborationen kommer en aritmetisk och logisk enhet (ALU) att konstrueras. Detta är ofta en grundsten i realiseringar som behöver någon form av aritmetik. Här kommer den att användas för att se hur två tal är relaterade. Du skall behärska talrepresentation, exempelvis skillnaden mellan `1000` och `0001`, samt 2-komplementräkning.

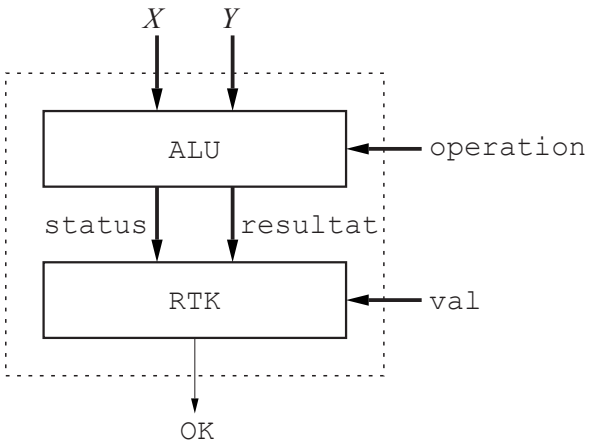
Följande kretsar finns till laborationen:

Antal	Krets	Beskrivning
6	7400	4 st 2 ingångars NAND
1	7404	6 st inverterare
2	7408	4 st 2 ingångars AND
2	7410	3 st 3 ingångars NAND
2	7420	2 st 4 ingångars NAND
5	7486	4 st 2 ingångars EXOR
1	HDSP-H101	Display

Kretsarnas placering på labkortet



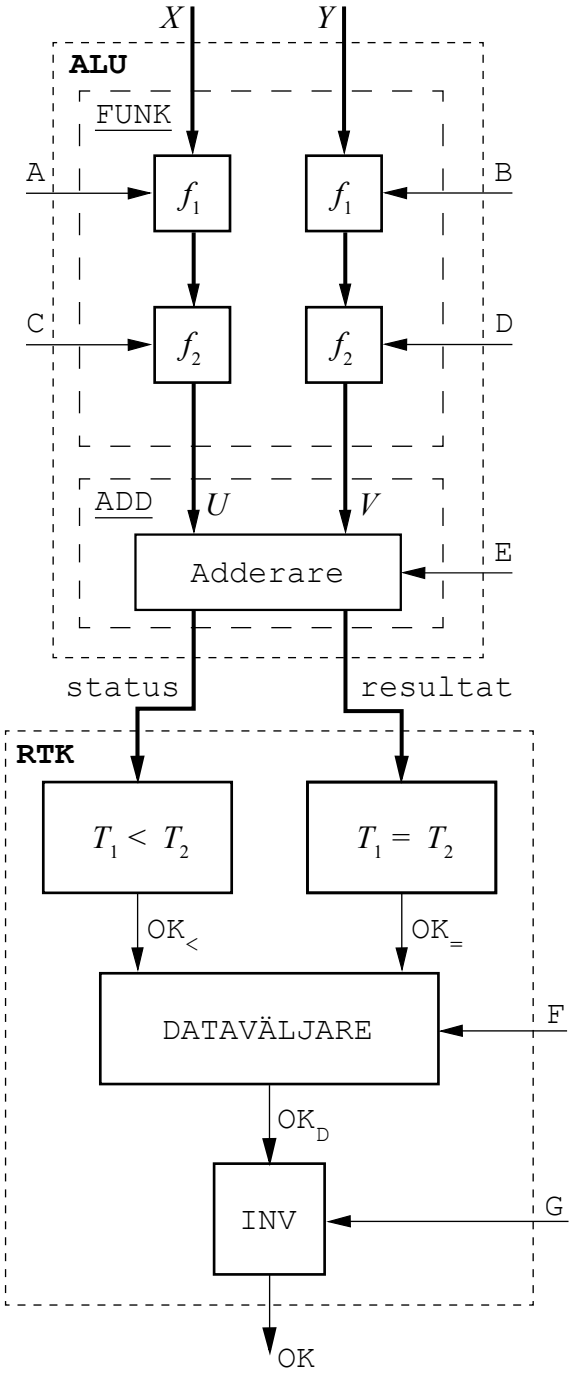
Under laborationen skall du konstruera ett nät som jämför de två fyrabitarstalen X och Y. Ett vanligt sätt att kontrollera när en given relation är uppfylld är att först bilda skillnaden mellan talen. Därefter tolkas resultatet och statussignalerna för att bilda svaret. Därför skall en enkel ALU (aritmetisk logisk enhet / Arithmetic and Logic Unit) konstrueras. Denna skall kunna utföra addition och subtraktion av fyrabitarstal. För att tolka resultatet från ALU:n skall ett byggblock `RTK` (resultattolk) konstrueras. Detta ger ut en bit, `OK`, som anger om relationen är sann eller falsk. I figur 2.1 visas ett blockschema över konstruktionen.



Figur 2.1: Principskiss av realiseringen av `OK`. Pilar i fetstil representerar flera binära signaler medan tunna pilar representerar en binär signal.

Styrsignalen `operation` bestämmer vilken funktion som skall utföras i blocket `ALU` medan styrsignalen `val` bestämmer hur resultatet skall tolkas. Styrsignalerna består av flera bitar, där följande gäller `operation = (A,B,C,D,E)` och `val = (F,G)`. Däremot är resultatet `OK` binärt (sant eller falskt). Det är anledningen till att pilarna för `operation` och `val` är tjockare än pilen för `OK`.

I figur 2.2 finns en något mer detaljerad och övergripande bild på konstruktionen.



Figur 2.2: Detaljerad bild av konstruktionen som skall realiseras under laboratioen.

Exempel 2.1
Antag att vi vill testa funktionen $X < Y$. Då sätts `operation` *så att* `ALU` *utför $X - Y$. Sedan sätts* `val` *så att* `RTK` *ger 1 (sant) om resultatet är negativt och 0 (falskt) annars.*

I de tre första laborationsuppgifterna skall blocket `ALU` konstrueras. Det delas i sin tur upp i två byggstenar, nämligen `ADD` som konstrueras i den första uppgiften och `FUNK` som konstrueras i den andra uppgiften, se figur 2.2. I tredje uppgiften modifieras `ADD` så att statussignalen ut från `ALU` blir komplett.

Uppgift 2.1.

Till att börja med skall det kombinatoriska nätet `ADD` konstrueras. Detta är en adderare för fyrabitars binärkodade heltal. Nätet har tre insignaler. Två av dem är fyrabitarstal, $U = (u_3, u_2, u_1, u_0)$ och $V = (v_3, v_2, v_1, v_0)$, och den tredje är ingående carry, eller minnessiffra, c_0. Styrsignalen `E` i figur 2.2 är just c_0.

`ADD` skall realiseras med fyra heladderare, HA.[4] En HA har tre insignaler, u_i, v_i och c_i och två utsignaler, z_i och c_{i+1}. Utsignalerna beskriver summan av (de binära) insignalerna,

$$2 \cdot c_{i+1} + z_i = u_i + v_i + c_i.$$

För att konstruera en additionsmodul för flera bitar kan ett antal HA kopplas i serie.

`ADD` har två utsignaler, `status` och `resultat` där `status` består av två bitar (mest signifikant bit, msb, och overflow) medan `resultat` är fyra bitar som representerar summan $U + V + c_0$,

$$\begin{aligned} \texttt{status} &= (z_3, \texttt{OF}) \\ \texttt{resultat} &= (z_3, z_2, z_1, z_0) = U + V + c_0 \mod 2^4 \end{aligned}$$

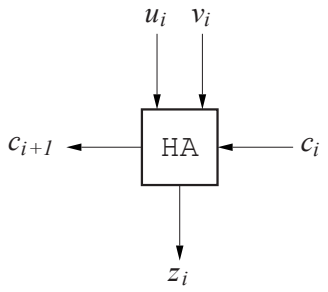
Overflow signalen, `OF`, skall vara ett om resultatet ligger utanför talområdet.

[4] I boken används den engelska termen FA, Full Adder. Eftersom modulen är en av de viktigaste grundstenarna för digitala konstruktioner har den även fått en svensk översättning.

Hemuppgift 2.1.1

Fyll i nedanstående funktionstabell för heladderaren, `HA`.

u_i	v_i	c_i	c_{i+1}	z_i
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



Ange lämpliga uttryck till ovanstående funktioner (se kursboken eller föreläsningsbilder).

$z_i =$

$c_{i+1} =$

Hemuppgift 2.1.2

Rita ett schema (på grindnivå) för byggblocket `ADD`. Använd inte AND grindar för att realisera byggblocket, dessa behövs i nästa uppgift.

Koppla upp ovanstående nät och kontrollera dess beteende. Använd vippströmbrytarna längst ner till vänster på kortet för att generera talen A och B. Ingående carry, c_0, ansluter du lämpligen till en logisk nolla eller etta (dessa finns i mitten på kopplingplattan). Koppla utsignalerna till lysdioder.

Vad blir resultatet, Z, av $(0101) + (0111)$? Ange också vad c_1, c_2, c_3 och c_4 är.

Svar: _____

Kalla på en laborationshandledare som skall kontrollera denna byggsten.

Slut på uppgift 2.1

Nätet i ovanstående uppgift skall behållas. Du skall nu modifiera det så att ett antal funktioner kan realiseras. För att realisera en given funktion skall en kombination av ett antal *styrsignaler* (`=operation`) ges till nätet.

Uppgift 2.2.

Nätet som är konstruerat realiserar funktionen $Z = X+Y$. Blocket `ALU` skall ha möjlighet att generera ytterligare funktioner, exempelvis $Z = X - Y$, $Z = Y - X$, $Z = X$, $Z = Y$ och $Z = 0$. Detta görs genom ett antal nya delfunktioner. Talen X och Y skall, var för sig, kunna nollställas och/eller inverteras. Till det behövs fyra styrsignaler, två för varje tal.

Hittills har kopplingen inte varit beroende av om talen är representerade som teckenbelopp eller 2-komplement men för att realisera subtraktion måste det bestämmas. Eftersom addition är enkel även för negativa tal i 2-komplementsrepresentation väljer vi den representationen.

Hemuppgift 2.2.1

För att beräkna $X - Y$ realiseras egentligen $X + (-Y)$. Hur kan ett fyrabitarstal $T = (t_3, t_2, t_1, t_0)$ negeras, dvs hur representeras $-T$ i 2-komplement?

$-T =$

Den första delfunktionen som skall konstrueras bestämmer om ett tal skall nollställas eller inte. Argumentet till funktionen består av en styrsignal $\texttt{styr}_1$ och fyrabitarstalet $T = (t_3, t_2, t_1, t_0)$,

$$f_1(\texttt{styr}_1, T) = \begin{cases} 0, & \text{om } \texttt{styr}_1 = 0 \\ T, & \text{om } \texttt{styr}_1 = 1 \end{cases}$$

Hemuppgift 2.2.2

Antag att argumentet till funktionen ges av $T = (t_3, t_2, t_1, t_0)$. Realisera delfunktionen $f_1(\texttt{styr}_1, T)$.

Det behövs ytterligare en delfunktion för att realisera FUNK. Den bestämmer om talet skall inverteras (positionsvis) eller inte,

$$f_2(\texttt{styr}_2, T) = \begin{cases} T, & \text{om } \texttt{styr}_2 = 0 \\ T' = (t_3', t_2', t_1', t_0'), & \text{om } \texttt{styr}_2 = 1 \end{cases}$$

Hemuppgift 2.2.3

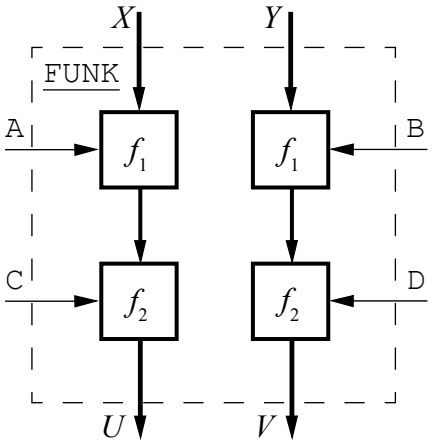
Antag att argumentet till funktionen ges av $T = (t_3, t_2, t_1, t_0)$. Realisera delfunktionen $f_2(\texttt{styr}_2, T)$.

Nätet kan nu realiseras enligt figur 2.3. Låt X vara argument till en funktion f_1. Motsvarande styrsignal betecknas med A. Utsignalen skall vara argument till en funktion f_2 vars styrsignal är C. Utsignalen från f_2 kan anslutas till U-ingången på ADD. Då blir

$$U = f_2(\texttt{C},\ f_1(\texttt{A}, X))$$

Låt på samma sätt Y vara argument till en funktion f_1 med styrsignal B. Utsignalen ges som argument till en funktion f_2 vars styrsignal är D. Denna utsignalen anslutas till ADD-nätets V-ingång. Då blir

$$V = f_2(\texttt{D},\ f_1(\texttt{B}, Y))$$



Figur 2.3: Skiss för hur byggblocket FUNK realiseras.

Exempel 2.2

För att beräkna funktionen $X - Y$ skall X oförändrad in på U-ingången på ALU. Det får vi om $\texttt{A} = 1$ och $\texttt{C} = 0$. Till V-ingången skall $-Y$ kopplas ($-Y = Y' + 1$), vilket vi får om $\texttt{B} = 1$, $\texttt{D} = 1$ och $\texttt{E} = 1$.

Hemuppgift 2.2.4

Vilka funktioner erhålles då styrsignalerna har nedanstående värden?

Funktion	A	B	C	D	E
$Z =$	1	0	0	1	1
$Z =$	0	0	1	1	1
$Z =$	1	1	0	0	1
$Z =$	0	0	1	0	1

Hemuppgift 2.2.5

Fyll i styrsignalerna `(A,B,C,D,E)` så att de specificerade funktionerna erhålles. Tre exempel visas, varav ett är förklarat i exempel 2.2.

Funktion	A	B	C	D	E
$Z = X$					
$Z = X + 1$					
$Z = X - 1$	1	0	0	1	0
$Z = Y$					
$Z = Y + 1$					
$Z = Y - 1$					
$Z = X + Y$					
$Z = X - Y$	1	1	0	1	1
$Z = Y - X$					
$Z = (0, 0, 0, 0)$	0	0	0	0	0
$Z = (1, 1, 1, 1)$					
$Z = (0, 0, 0, 1)$					

Koppla upp byggblocket `FUNK` och koppla ihop dess utsignaler med `ADD`. Anslut styrsignalerna `(A,B,C,D,E)` till vippströmbrytare (till höger om de tidigare använda brytarna). Kontrollera ett antal av funktionerna i hemuppgift 2.2.5.

Kalla på en laborationshandledare och demonstrera ditt nät.

Slut på uppgift 2.2

Ovanstående nät klarar av att addera och subtrahera. Det skall nu kompletteras med statussignalen *overflow*, `OF`, för att signalen `OK` skall kunna realiseras.

Uppgift 2.3.

Utsignalen `status` från blocket `ADD` i figur 2.2 består av msb z_3 och overflow `OF`.

Hemuppgift 2.3.1

Fyll värdet för z_3 och c_4 (utgående carry) i nedanstående tabell. Fyll därefter i värdet för signalen `OF`, tänkt på att Z ska tolkas som 2-komplement. *Tips: Overflow inträffar då två tal med samma tecken adderas och resultatet har motsatt tecken!*

u_3	v_3	c_3	c_4	z_3	OF
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Ange ett lämpligt uttryck för funktionen `OF`. *Tips: Jämför värdet hos signalerna c_3 och c_4 då overflow inträffar!*

`OF` $=$

Realisera funktionen `OF`.

De utsignaler som nu finns tillgängliga är svaret från ALU:n, Z, utgående carry c_4 samt overflow-signalen `OF`. Med hjälp av dessa och två nya styrsignaler, `F` och `G`, skall den resterande logiken realiseras. Tanken är att vi med styrsignalerna $(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G})$ ställer in kretsen så att en av följande relationer väljes

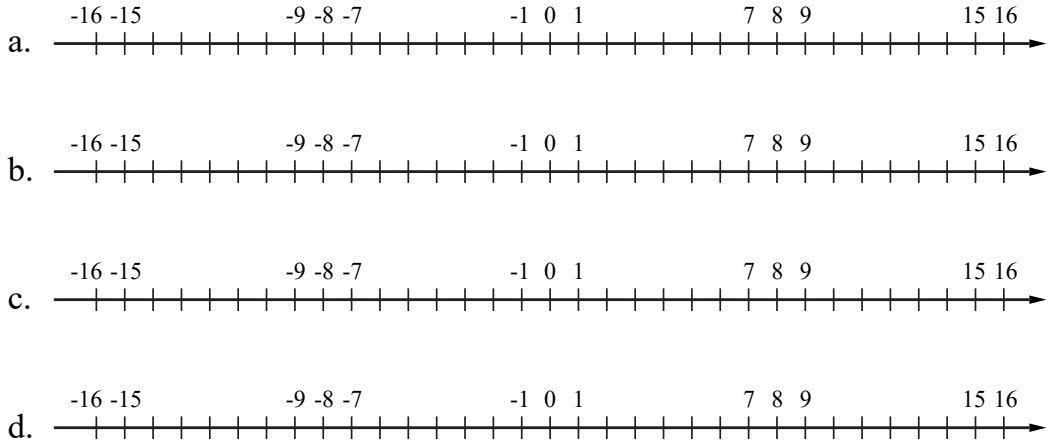
$$\begin{array}{ll} X < Y & X \geq Y \\ Y < X & Y \geq X \\ X = Y & X \neq Y \end{array}$$

Först används $(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E})$ för att bilda skillnaden mellan de två talen, antingen som $X - Y$ eller som $Y - X$. Signalen `OK` kan nu beräknas ur resultatet och styrsignalerna `F` och `G`.

För att se hur `OK` kan beräknas ur resultatet skall vi titta närmare på subtraktionen mellan två tal och i vilka intervall resultatet ligger. Före nästa hemuppgift är det vettigt att repetera talrepresentation och 2-komplement i kursboken. Speciellt kan det vara bra att förstå sambandet mellan tallinjen i uppgiften och talcirkeln för moduloberäkning.

Hemuppgift 2.3.2

Betrakta nedanstående tallinje.



Du skall nu i ett antal steg markera olika intervall och värden i figuren ovan. Resultaten skall sedan användas i nästa uppgift.

a. Antag att du har två tal T_1 och T_2 som kan anta värden i intervallet $\{-8, -7, \ldots, 7\}$. I vilket intervall finner du resultatet av operationen $Z = T_1 - T_2$ (resultatet är inte begränsat av 4 bitar)? Markera intervallet i figuren ovan.

b. Antag att subtraktionen $Z = T_1 - T_2$ är utförd och att du vill kontrollera om relationen $T_1 < T_2$ är uppfylld. I vilket intervall kommer resultatet att finnas i då relationen sann? Markera under tallinjen.

c. Om subtraktionen utförs med en 4 bitars ALU kommer resultatet i vissa fall hamna utanför talområdet $\{-8, -7, \ldots, 7\}$, dvs det blir en overflow. Markera intervallen då overflow-signalen `OF` är ett respektive noll. Märk även intervallen med etta respektive nolla så att det tydligt framgår vilket intervall som är vilket.

d. Resultatet av subtraktionen i ALU:n ges av $Z = T_1 - T_2$. Där Z representerar resultatet med 4 bitar (2-komplement) $Z = (z_3, z_2, z_1, z_0)$. Markera intervallen då den mest signifikanta biten z_3 är ett respektive noll. Märk även intervallen med etta respektive nolla så att det tydligt framgår vilket intervall som är vilket.

Föregående hemuppgift illustrerar hur en modul kan realiseras med vars hjälp du kan testa när $X < Y$ eller när $X > Y$. Valet av relation bestäms av ALU-operationen och påverkar inte den nya modulen. Det är också möjligt att realisera $X \geq Y$ och $Y \geq X$ genom att betrakta komplementet. För att realisera dessa funktioner behövs ett block som realiserar funktionen

$$\texttt{OK}_< = \begin{cases} 1 \text{ , om } T_1 < T_2 \\ 0 \text{ , annars} \end{cases}$$

Hemuppgift 2.3.3

Antag att ALU:n utför operationen $Z = T_1 - T_2$. Utgå från markeringarna du gjorde föregående uppgift och fyll i nedanstående tabell. $T_1 < T_2$ ska vara ett då relationen är sann, annars noll.

OF	z_3	$T_1 < T_2$
0	0	
0	1	
1	0	
1	1	

Realisera funktionen

$\texttt{OK}_< =$

För att realisera funktionerna $X = Y$ och $X \neq Y$ behövs ett block som realiserar funktionen

$$\texttt{OK}_= = \begin{cases} 1, & \texttt{T}_1 = \texttt{T}_2 \\ 0, & \text{annars} \end{cases}$$

Hemuppgift 2.3.4

Antag att ALU:n utför operationen $Z = T_1 - T_2$. Hur kan funktionen ovan realiseras?

$\texttt{OK}_= =$

Med hjälp av resultatet från de två ovanstående funktionerna kan nu alla relationerna på sidan 24 kontrolleras med hjälp av blockschemat i figur 2.4. Blocket DATAVÄLJARE

släpper igenom en av ingångarna till utgången enligt

$$\texttt{DATAVÄLJARE} = \begin{cases} \texttt{OK}_{<}, & \texttt{F} = 0 \\ \texttt{OK}_{=}, & \texttt{F} = 1 \end{cases}$$

Hemuppgift 2.3.5

Hur kan blocket `DATAVÄLJARE` realiseras (rita)?

För att invertera utgången på `DATAVÄLJARE` behövs ett block `INV` som inverterar insignalen $\texttt{OK}_\texttt{D}$ om styrsignalen `G` är 1,

$$\texttt{INV} = \begin{cases} \texttt{OK}_\texttt{D}, & \texttt{G} = 0 \\ \texttt{OK}_\texttt{D}', & \texttt{G} = 1 \end{cases}$$

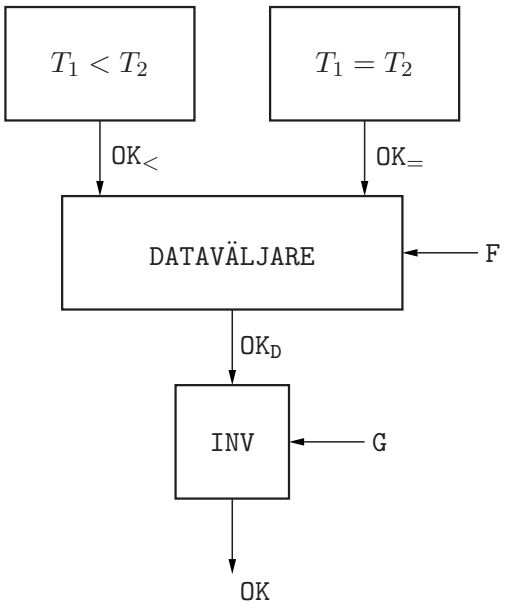
Hemuppgift 2.3.6

Hur kan blocket `INV` realiseras?

Exempel 2.3

För att kontrollera relationen $X \leq Y$ utförs subtraktionen $Y - X$ i ALU:n. Då finns resultatet för relationen $Y < X$ i $\texttt{OK}_{<}$. Denna skall släppas igenom `DATAVÄLJARE` genom att $\texttt{F} = 0$. Sedan skall signalen inverteras för att få den önskade relationen, så vi sätter $\texttt{G} = 1$. Den fullständiga uppsättningen styrsignaler blir alltså

$$(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}) = (1, 1, 1, 0, 1, 0, 1)$$



Figur 2.4: Blockschema över modulen `RTK`.

Hemuppgift 2.3.7

Fyll i styrsignalerna i nedanstående tabell så att de givna relationerna besvaras. Signalerna från exempel 2.3 är ifyllda.

Relation	A	B	C	D	E	F	G
$X = Y$							
$X \neq Y$							
$X < Y$							
$X > Y$							
$X \leq Y$	1	1	1	0	1	0	1
$X \geq Y$							

Koppla upp modulen i figur 2.4 och anslut `F` och `G` till vippströmbrytarna bland de övriga styrsignalerna. Testa olika argument X och Y för de olika relationerna för att se att det fungerar.

Kalla på en laborationshandledare som skall kontrollera din lösning.

Slut på uppgift 2.3

Nästa uppgift är fristående från de tidigare i denna laborationen. Den går ut på att realisera en drivkrets, liknande den som användes i laboration 1, för att visa binärt repre-

senterade tal på en display.

Uppgift 2.4.

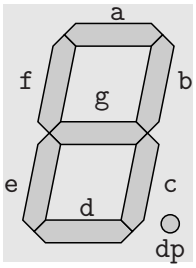
I denna uppgift skall ett kombinatoriskt nät konstrueras. Nätet har tre ingångar (den binära representationen) och sju utgångar (en för varje segment på displayen). Talen som skall visas på displayen är decimala, $0 \leq X \leq 5$, och kodade med tre bitars naturlig binärkod enligt

$$X = x_2 \cdot 2^2 + x_1 \cdot 2 + x_0.$$

Det kallas att talen är *NBCD*-kodade (Natural Binary Coded Decimal).

Displayen är uppbyggd med sju segment och en punkt enligt figur 2.4. Punkten skall inte användas i denna uppgift. Segmenten är märkta `a`, `b`, `c`, `d`, `e`, `f` och `g`.

Nere till höger på kretskortet finns de ben med vars hjälp du skall styra displayen.



Figur 2.5: En display med sju segment `a`–`g` och en punkt `dp`.

Hemuppgift 2.4.1

Fyll i nedanstående tabell. Börja med att markera hur siffrorna skall se ut. Ett exempel är redan ifyllt i tabellen. *Observera att displayen bara behöver kunna visa talen 0–5.*

X	x_2	x_1	x_0	utseende	a	b	c	d	e	f	g
0	0	0	0		1	1	1	1	1	1	0
1	0	0	1								
2	0	1	0								
3	0	1	1								
4	1	0	0								
5	1	0	1								
6	1	1	0								
7	1	1	1								

Hemuppgift 2.4.2

Beräkna minimala (var för sig) Booleska uttryck för de sju utsignalerna `a` till `g`.

`a` =

`b` =

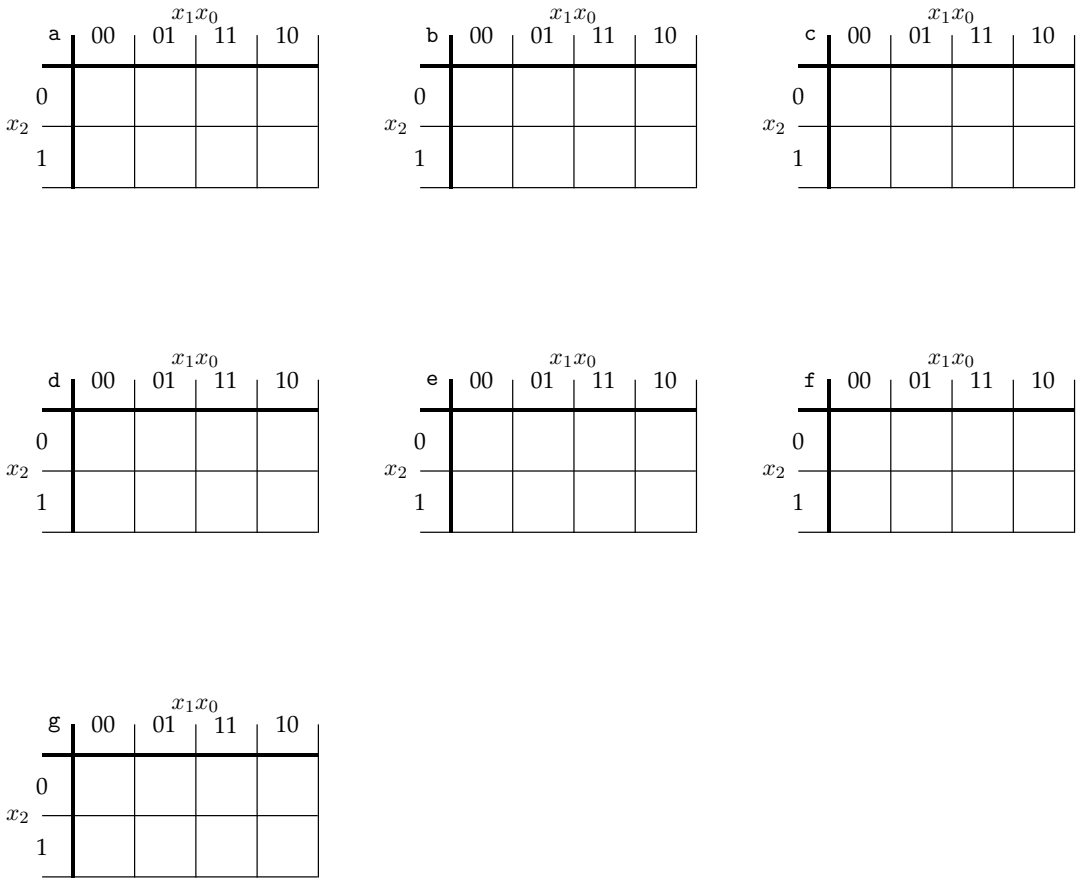
`c` =

`d` =

`e` =

`f` =

`g` =



Koppla upp ovanstående nät och testa det.

Vad visar displayen om ni ställer in ett tal som är utanför intervallet noll till fem? Varför?

Svar: _____

Slut på uppgift 2.4

LABORATION 3

LATCHAR OCH VIPPOR

Laborationens syfte

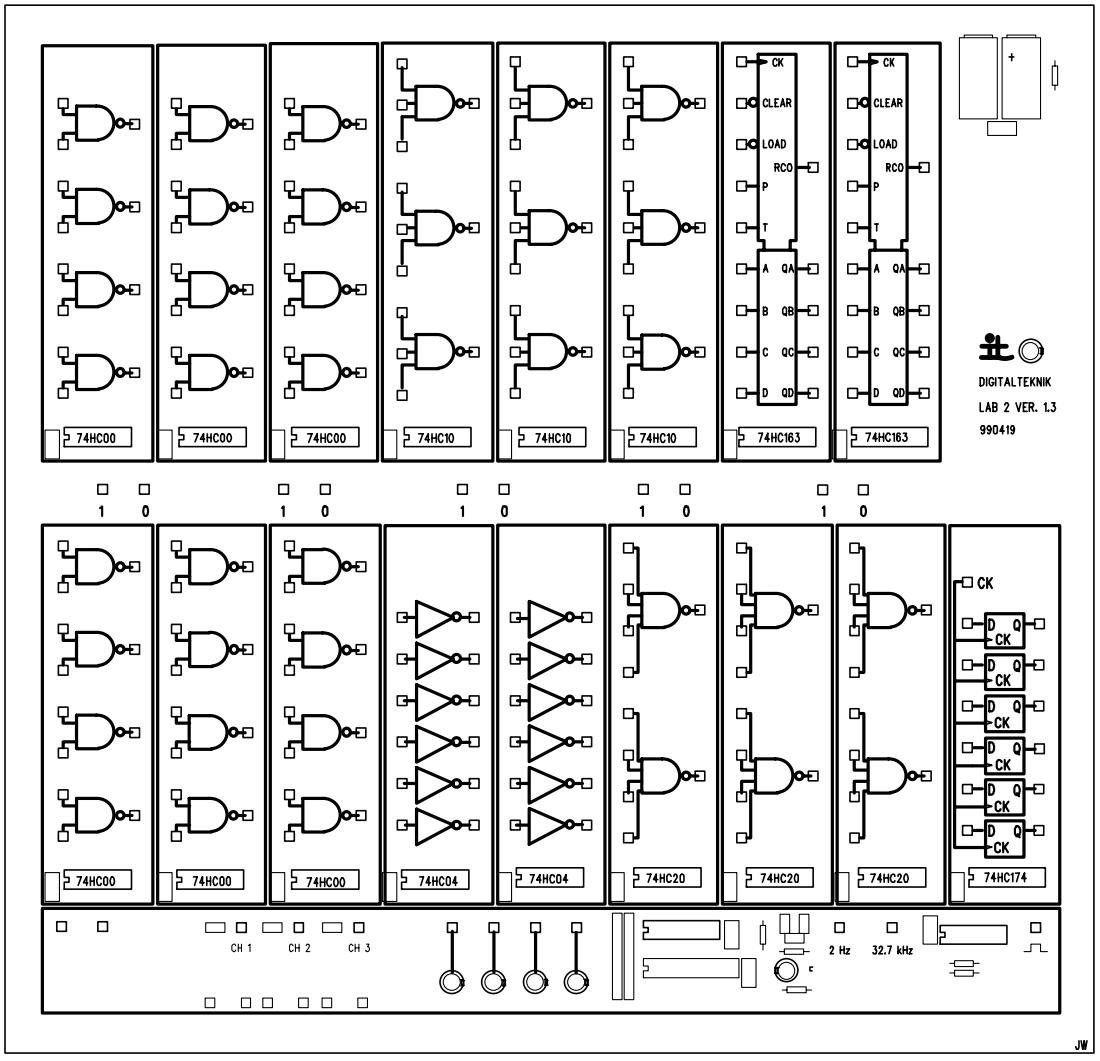
Denna laborationen handlar om sekvensnät. Det är återigen samspelet mellan det kombinatoriska nät och fördröjningselement, *D*-vippor, som är det centrala. Till att börja med skall principerna för en *D*-vippa belysas. Det sker genom att dess grundsten, latchen, först realiseras. Två latchar byggs sedan ihop till en vippa. Klocksignalen spelar en viktig roll i dessa nät och vi skall även titta närmre på när vippan uppdateras.

Efter denna laboration skall skillnaden mellan *Moore-* och *Mealy-maskiner* vara förtydligad. Andrar kunskaper som kommer att erfaras är *tillståndsminimering* och *tillståndskodning*. Sista delen av laborationen kommer handla om en *räknare*.

Följande kretsar finns till laborationen:

Antal	Krets	Beskrivning
6	7400	4 st 2 ingångars NAND
2	7404	6 st inverterare
3	7410	3 st 3 ingångars NAND
3	7420	2 st 4 ingångars NAND
1	74174	6 st D-vippor
2	74163	Synkron 4 bitars räknare

Kretsarnas placering på labkortet

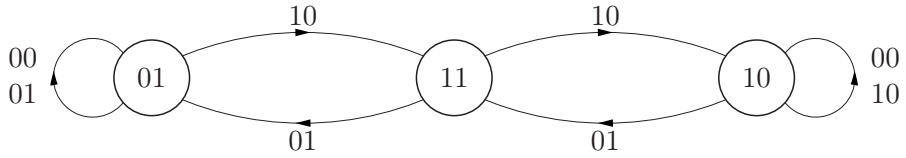


I ett sekvensnät är D-vippor av central betydelse, det är här tillstånden lagras. Därför inleds laborationen med en undersökning av hur de kan realiseras. Vippor är i själva verket även de sekvensnät, de är dock asynkrona. Det bör noteras att en vippa kan realiseras på olika vis. Vipporna som kommer att konstrueras under denna laboration skiljer sig från de som beskrivs i kursboken.

Uppgift 3.1. Latch

Till att börja med behövs någon typ av minne. I denna realisering skall vi använda två ingångar, x_1 och x_2, och två utgångar, f_1 och f_2. Då insignalerna är $(x_1, x_2) = (1, 0)$ eller $(x_1, x_2) = (0, 1)$ skall utsignalerna sättas till detsamma. Då insignalen ändras till $(x_1, x_2) = (0, 0)$ skall värdet på utgångarna inte ändras.

Det innebär att vi behöver en graf för ett asynkront (utan klocksignal) sekvensnät där två tillstånd är stabila. Vi realiserar det som en Moore-maskin där tillståndsvariablerna fungerar direkt som utsignaler. Det är således tillstånden 10 och 01 som är de stabila tillstånden. För att få en kapplöpningsfri tillståndskodning måste grafen kompletteras med ett tillstånd, här väljer vi att koda det med 11. I figur 3.1 visas grafen där tillståndsvariablerna är f_1 och f_2.



Figur 3.1: Graf för en minneskrets.

Hemuppgift 3.1.1

Visa att grafen i figur 3.1 kan realiseras med funktionerna

$$f_1{}^+ = (x_1' \wedge f_2)'$$
$$f_2{}^+ = (x_2' \wedge f_1)'$$

Eftersom det finns beroende i form av återkopplingar är det inte ett kombinatoriskt nät utan ett asynkront sekvensnät.

Hemuppgift 3.1.2

Ovanstående nät kallas för ett *bistabilt element* eller en *latch*. Rita nätet.

Bygg nätet i hemuppgift 3.1.2 och fyll i nedanstående tabell.

x_1	x_2	f_1	f_2
0	0		
0	1		
1	0		
1	1		

Kan tabellen fyllas i entydigt?

Svar: _____

Förbind insignalerna, dvs $x = x_1 = x_2$. Vad händer när insignalen x ändras?

Svar: _____

Går det att förutse vad utsignalerna blir när $x : 1 \longrightarrow 0$? Prova många gånger och se om det alltid blir samma.

Svar: _____

Antag att en godtycklig insignalskombination (insignalerna är inte förbundna med varandra) kan följas av $(0, 0)$. Vilken insignalskombination bör undvikas om du skall kunna förutsäga utsignalerna efter återgång till $(x_1, x_2) = (0, 0)$?

Svar: _____

Kalla på en labhandledare och motivera dina svar.

Slut på uppgift 3.1

En egenhet hos den nu konstruerade latchen är att insignalerna propagerar genom nätet ohindrat. Detta kan leda till problem i synkrona sekvensnät där systemet skall uppdateras vid speciella tillfällen. Därför skall kopplingen kompletteras med en styrsignal (denna kommer senare att ersättas utav signalen från klockan). Därefter seriekopplas två latchar för att bilda en vippa. En vippa är ett minneselement där utsignalen endast uppdateras vid en specifik tidpunkt, denna koppling är således ett synkront sekvensnät.

Uppgift 3.2. Styrd latch

Till att börja med skall en styrsignal, ϕ, läggas till så att ingångarna kan stängas av. I grafen i figur 3.1 får vi då tre insignaler. Så länge ϕ är noll är ingångarna avstängda och endast signalen $(0, 0)$ kommer fram till minneselementet. Ett sätt att åstadkomma detta är att låta ingångarna gå via AND-grindar tillsammans med ϕ. Det blir samma sak som att byta ut inverterarna på ingångarna i föregående koppling mot 2-ingångars NAND-grindar där också ϕ är insignal.

Hemuppgift 3.2.1

Rita det nät som ges av funktionerna:

$$f_1 = ((x_1 \wedge \phi)' \wedge f_2)'$$
$$f_2 = ((x_2 \wedge \phi)' \wedge f_1)'$$

Koppla upp nätet i hemuppgift 3.2.1 och prova det.

Slut på uppgift 3.2

Fortfarande är det problem då $\phi = 1$ eftersom ingångarna påverkar utgångarna direkt. Problemet kan lösas genom att två identiska latchar kopplas seriellt enligt ett MASTER-SLAVE-förhållande. När den första (MASTER) är låst släpper den andra (SLAVE) igenom dess värde, och när den första är öppen har den andra fryst värdet på utgången.

Uppgift 3.3. Vippa

För att bygga en vippa behövs två identiska latchar. Koppla därför upp ytterligare en styrd latch enligt förra uppgiften. Kontrollera att den uppför sig som den skall.

Kalla de två latcharna L_1 och L_2. De skall nu kopplas enligt ett MASTER-SLAVE-förhållande där L_1 är MASTER-latch och L_2 SLAVE-latch. Anslut utgångarna från L_1 till ingångarna på L_2, dvs $x_i^{(L_2)} = f_i^{(L_1)}, i = 1, 2$. Då styrsignalen till MASTER-latchen är $\phi_1 = 0$ och L_1 är låst, är SLAVE-latchen öppen, $\phi_2 = 1$, och skickar vidare värdet till utgången. Vid omslag slår först SLAVE-latchen om och låser utsignalen, och sedan öppnar MASTER-latchen. Eftersom L_2 har låst utsignalen kommer denna inte att ändras förrän nästa omslag då L_1 först låser sin utgång varefter L_2 öppnar och släpper igenom den. Det är alltså bara

insignalen vid tiden då ϕ_2 ändras från 0 till 1 som utsignalen kan ändras. Vid alla andra tillfällen är utsignalen oförändrad oavsett vad insignalen är.

För att lösa detta behöver vi konstruera en styrkrets som genererar ϕ_1 och ϕ_2 enligt ovan. Vi kan tänka oss det som en asynkron tillståndsmaskin en insignal clk, och två tillståndsvariabler (ϕ_1, ϕ_2). Det finns två stabila tillstånd, $(0, 1)$ för insignal $\mathrm{clk} = 0$ och $(1, 0)$ för insignal $\mathrm{clk} = 1$. Vid omslag måste ett tredje tillstånd användas eftersom det är en asynkron realisering. För att undvika direktkoppling mellan in- och utsignal välj $(0, 0)$.

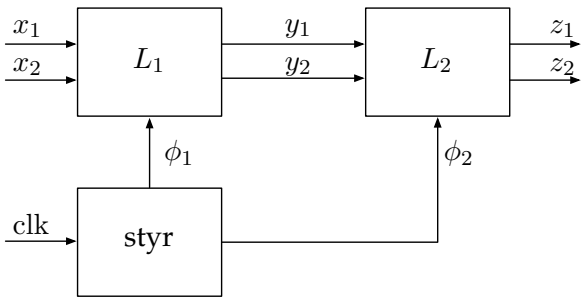
Hemuppgift 3.3.1

Konstruera en graf för styrkretsen enligt beskrivningen ovan. Realisera maskinen.

Koppla upp och testa styrkretsen i föregående hemuppgift. Kontrollera de stabila tillstånden

$$(\phi_1, \phi_2) = \begin{cases} 0, 1, & \mathrm{clk} = 0 \\ 1, 0, & \mathrm{clk} = 1 \end{cases}$$

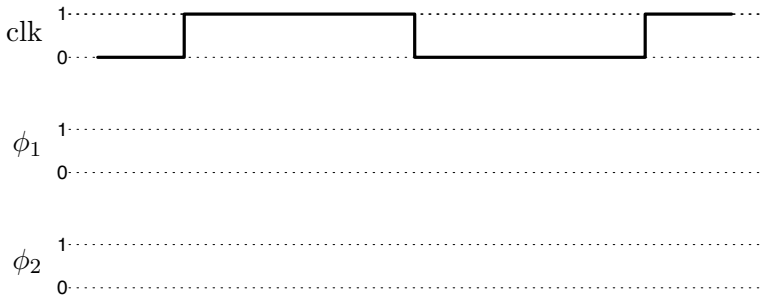
Koppla ihop styrkretsen med de två kaskadkopplade latcharna enligt figur 3.2.



Figur 3.2: Blockdiagram för vippa med MASTER-SLAVE-struktur.

Hemuppgift 3.3.2

Rita ett tidsdiagram över signalerna clk, ϕ_1 och ϕ_2 i figur 3.2 och beskriv vad som händer i kretsen vid de olika händelserna.



Verifiera tidsdiagrammet i föregående hemuppgift. Vippan i realiseringen kallas för en SR-vippa (från S=Set och R=Reset) och har följande beteende

x_1	x_2	z_1	z_2
0	0	z_{1_0}	z_{2_0}
0	1	0	1
1	0	1	0

Där z_{1_0} och z_{2_0} betyder att signalerna behåller sina tidigare värden. Insignalen $(x_1, x_2) = (1, 1)$ bör som bekant undvikas då det inte går att förutse vilka värde utsignalerna antar då $(x_1, x_2) \to (0, 0)$.

En vippa där uppdateringen av utsignalerna initieras av att klocksignalen ändras (exempelvis från 0 till 1), på flanken mellan två signalsvärden, sägs vara *flanktriggad*. Om flanken är från 0 till 1 (⎍) säger vi att den är *positivt flanktriggad*. På motsvarande sätt finns det *negativt flanktriggade* vippor som uppdaterar på omslag från 1 till 0 (⏋). I datablad markeras att vippor är flanktriggade enligt figur 3.3.

Figur 3.3: Symbolmarkering för flanktriggade vippor.

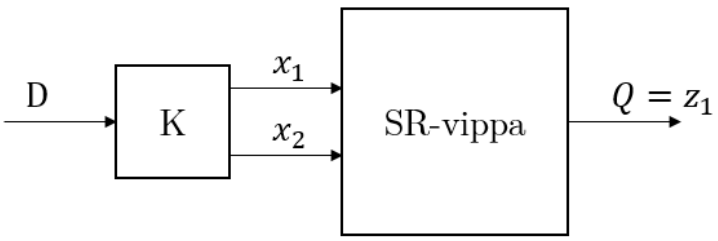
Är vippan som du konstruerat positivt eller negativt flanktriggad? *Tips: använd den manuella klockan för att kontrollera när utsignalerna ändras.*

Svar: _____

D-vippan som du använt i tidigare laborationer har, till skillnad från SR-vippan, bara en insignal `D` och en utsignal `Q` med följande beteende

D	Q
0	0
1	1

Du ska nu modifiera SR-vippan så att den beter sig som en D-vippa genom att lägga till blocket `K` enligt figur 3.4



Figur 3.4: Modifiering av SR-vippan så att det blir en D-vippa.

Hemuppgift 3.3.3

Realisera blocket `K` så att konstruktionen beter sig som en D-vippa.

Ändra kopplingen så den blir en *D*-vippa.

Kalla på en labhandledare som skall kontrollera ditt nät.

Slut på uppgift 3.3

Den första delen av laborationen är nu färdig. Metoden att använda två latchar efter varandra (MASTER-SLAVE) är typisk för hur en vippa realiseras. Däremot kan realiseringen av latcharna vara olika.

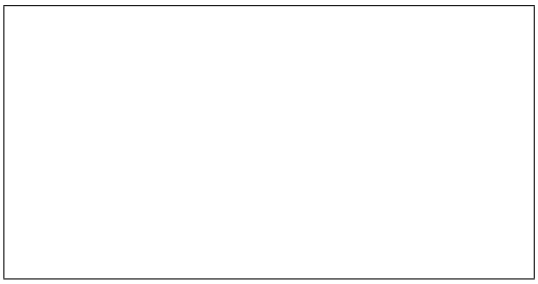
Vi skall nu gå över till att använda *D*-element för att spara tillståndet i en realisering.

Observera att instruktioner till följande hemuppgift lämnas vid laboration 2. Om du inte fått instruktioner se till att kontakta laborationsansvarig i god tid före labben, se kurshemsida.

Uppgift 3.4. Räknare

Hemuppgift 3.4.1

Konstruera en synkron modulo räknare enligt nedanstående specifikation (som tilldelas vid laboration 2). Endast D-vippor, NAND-grindar och NOT-grindar får användas. Insignalerna r_1 och r_2 anger räknarens beteende enligt nedanstående tabell.



Observera att räknarens tillstånd kan kodas godtyckligt, men olika kodning kan ge stor skillnad i storleken på nätet. Oavsett kodning är det tillståndskoden som skall ges som utsignal.

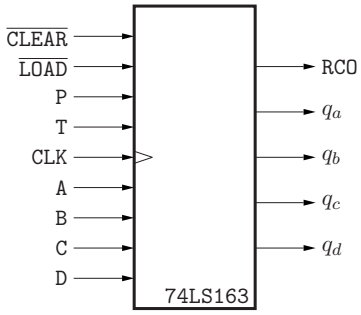
Koppla upp räknaren och kontrollera dess beteende. När du anser att den fungerar skall

en labhandledare kontrollera den.

Slut på uppgift 3.4

Uppgift 3.5. Standardräknare

I denna uppgift skall den integrerade kretsen `74LS163` användas. Kretsen är en synkron räknare, dvs all uppdatering av tillstånden sker på klocksignalens (`CLK`) positiva flank. Räknaren har $2^4 = 16$ tillstånd. Varje tillstånd, s_j, ges av fyrtippeln $Q = (q_d, q_c, q_b, q_a)$, där q_a är minst signifikant. Q är binärkoden av tillståndets index j. I figur 3.5 visas de in- och utgångar som är tillgängliga på räknaren.



Figur 3.5: Ingångar och utgångar på räknaren `74LS163`.

Räknaren påverkas av insignalvektorn ($\overline{\texttt{CLEAR}}, \overline{\texttt{LOAD}}, \texttt{P}, \texttt{T}, \texttt{D}, \texttt{C}, \texttt{B}, \texttt{A}$) enligt nedanstående tabell:

`(0,-,-,-,-,-,-,-)`	*nollställning,* $s^+ = s_0$.	`0-------` $s_j \rightarrow s_0$
`(1,0,-,-,d,c,b,a)`	*parallell laddning,* $s^+ = s_{(dcba)_2}$	`10--dcba` $s_j \rightarrow s_{\texttt{dcba}}$
`(1,1,1,0,-,-,-,-)`	*oförändrat tillstånd,* $s^+ = s$.	`11-0----` `110-----` s_j
`(1,1,1,1,-,-,-,-)`	*plus ett,* $s^+ = s_{(j+1 \mod 16)}$ då $s = s_j$	`1111----` $s_j \rightarrow s_{j+1}$

Förutom att tillståndsvariablerna är tillgängliga så finns ytterligare en utsignal `RCO` (Ripple Carry Output). Den ges av följande samband:

$$\texttt{RCO} = q_d \wedge q_c \wedge q_b \wedge q_a \wedge \texttt{T}.$$

`RCO` blir alltså 1 då räknaren skall slå om från 15 till 0 om $\texttt{T} = 1$. Det kan utnyttjas för att konstruera räknare med andra längder än 16.

Hemuppgift 3.5.1

Låt räknarens insignal vara `(1,RCO',1,1,d,c,b,a)` där datasignalerna `a,b,c,d` kommer från ett binärkodat tal enligt `tal`=(`d,c,b,a`).

Eftersom räknaren inte längre börjar om från noll är det nu en modulo-X räknare. Uttryck X med hjälp av det specificerade talet `tal`.

$X = f(\texttt{tal}) =$

Hemuppgift 3.5.2

Realisera en modulo-10 och en modulo-2 räknare, enligt föregående hemuppgift.

Antag att räknarna klockas med 2Hz klockan. Hur skall de kopplas för att få två utsignaler med frekvenserna 1 respektive 0.1 Hz?

Rita ett schema över räknarnas sammankoppling. *Observera att systemet skall vara synkront, dvs båda räknarna skall använda samma klocka.*

Koppla upp ovanstående nät och kontrollera dess beteende.

Kalla på en labhandledare och förklara vad som sker.

Slut på uppgift 3.5

LABORATORY LESSON 4

STOP WATCH

Purpose of the laboratory lesson

In this course fundamental digital design techniques like minimization of Boolean equations and application of Karnaugh-maps were introduced. When designing complex circuits built out of million of gates, however, these techniques are not practical. For such design an HDL (Hardware Description Language), e.g., VHDL or Verilog is applied to describe the circuits on a higher-level. On first glance these languages seem similar to C and Java. However, the inherent difference is that HDLs do not show a sequential flow but rather a concurrent flow.

In this lab you are going to use VHDL to implement a stop watch, simulate it and finally, prototype it on an FPGA board. The goal is to get an insight view on how complex designs are realized in industry and see how a problem can be divided into smaller pieces.

Files used in the laboratory

File name	Testbench	Description
nbcd.vhd	nbcd_tb.vhd	Binary to Natural Binary Coded Decimal
counter.vhd	-	Counter
edge_det.vhd	edge_det_tb.vhd	Edge detector
fsm.vhd	fsm_tb.vhd	Stop watch control
seven_segment.vhd	-	Seven segments driver
stop_watch.vhd	-	Stop watch top level design
stop_watch_sim.vhd	stop_watch_tb.vhd	Same as above but without debouncers at inputs
stop_watch_package.vhd	-	Debouncer and ADD3 components
top.xdc	-	Defines how signal at the top level of the design are connected to external components

Nexys4 FPGA development board layout

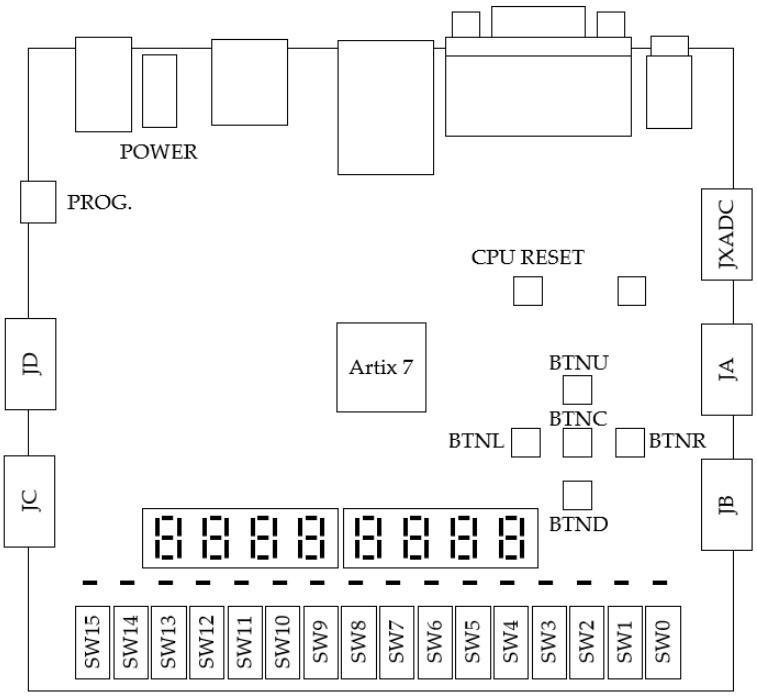


Figure 4.1: Switches `SW0` through `SW15` will produce a logic high when in their upper position. The cross arranged buttons, `BTNx`, will generate a logic high when pressed otherwise a logic low.

Home problem 4.0.1

During the lab you will use VHDL to describe your circuit. To implement the circuit on an FPGA you will use the software Vivado, which is Xilinx's development environment for FPGAs.

In appendix B a short tutorial on Vivado is given with an example project. Make sure to run through the tutorial before the lab.

The files you need during the lab are also available on the course homepage. Have a look at them before the lab.

Specification

During this lab you are going to develop a stop watch. The interface to users consist of three buttons, `start`, `stop` and `n_rst` as well as a seven-segment display.

When powering up the device the display should show 00.00, where the digits before the decimal point are seconds and the digits after are tenths and hundredths of a second. This means that the maximum count of the stop watch is 99.99 seconds. If the stop watch reaches this value it should wrap around and continue counting from zero.

In the initial state, i.e. after powering up or reset, the start button starts the stop watch and the current value is displayed on the display whereas the stop button is not supposed to have any function.

While the stop watch is running a one time press of the stop button pauses the stop watch and a second press resets the counted value; bringing the stop watch back to its initial state.

Pressing the start button during the pause state of the stop watch should bring the stop watch back to running mode, continuing counting at the paused value.

Whenever the reset signal `n_rst` goes low, the system shall be brought back and retained in its initial state as long as the reset signal is low. The reset should be asynchronous, i.e. reset should occur directly when `n_rst` is low without waiting for the next clock pulse.

The stop watch will be prototyped on a Nexys4 FPGA development board. Button `BTNR` is used as `stop` button, `BTNL` as `start` button and switch `SW0` will be used as reset signal `n_rst`, see Figure 4.1.

Block Diagram

To ease the development of complex designs, the design is split up in to a number of smaller, less complex blocks. This approach is called "Divide and Conquer". This partitioning is done below for you, but it would be a good exercise for you to think about this before continuing reading and to compare your result with the given design later on. The following things should be considered while doing this:

- How many inputs does the design have?
- What different functionalities are required and how can they be split into different blocks?
- What interfaces are necessary between the blocks to communicate?

The design will have two inputs (except for the reset and clock), a start and a stop button. Due to the mechanical nature of these, pressing them will not lead to an immediate stable signal but the signal will bounce; therefore, a debouncer is required. Figure 4.2 shows this behaviour.

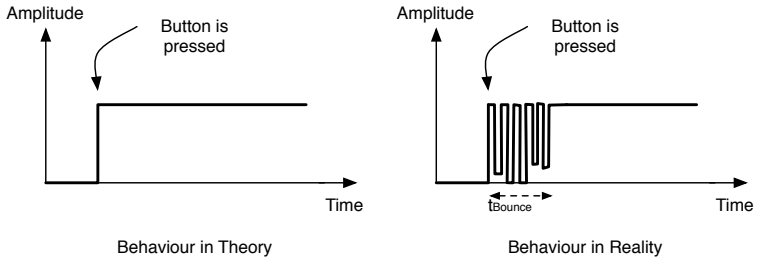


Figure 4.2: Behaviour of a mechanical switch: (left) in theory, (right) in reality.

Moreover, the FPGA will run on a high clock frequency. Hence, a single button press of the user will lead to an active signal at the input for many thousands or even millions of clock cycles. Thus, an edge detection circuit is needed to ensure that each push of a button only translates to a signal being active for one clock cycle.

To control the different states of the stop watch, an FSM (Finite State Machine) has to be employed in the design and subsequently this FSM has to trigger a counter that counts the time. This binary value has to be decoded into a representation matching the inputs of the seven-segment driver. The seven-segment driver will be provided and explained in detail later. Figure 4.3 shows the overall block diagram of the stop watch.

To ease the first steps a basic structure, i.e. a skeleton, of the whole design is prepared for you. The files needed during the lab are located in the folder **S:\Courses\EIT\EITF65\StopWatch**. It contains two sub-folders **vhdl** and **testbench**, the first contains all VHDL files that defines the stop watch, the second contains all testbenches needed during simulation.

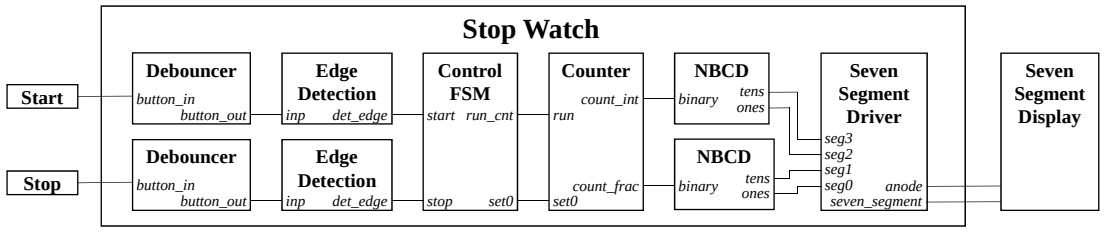


Figure 4.3: Block schematic of the stop watch. Clock and reset signals have been omitted for simplicity, however, keep in mind that all sequential circuits need to be reset using the system reset signal.

Throughout the lab all clocked gates should be sensitive to the **rising edge** of the clock.

Keep in mind while you are designing your circuits that different processes in a VHDL file run concurrently to each other and are triggered by their sensitivity list. Statements outside a process within an architecture are implicit processes which also run concurrently and are triggered by their operands.

Defining the Interfaces

After sketching a block diagram the interfaces required for communication between the blocks need to be determined. And the exact functionality of each block needs to be defined.

The debouncer circuit will only wait until the input button has a stable value for about 20 ms. Hence, it is a counter having the button as input `button_in` and a delayed output `button_out`. This block will be provided for you and is part of the package **stop_watch_package.vhd**.

Laboratory problem 4.1. Edge Detection

The edge detector circuit detects the rising edge of the debouncer output `button_out` at its input `inp` and acknowledges this with a one cycle active output signal at `det_edge`.

Home problem 4.1.1

Draw a sequential circuit that detects the rising edge of a signal using standard gates, e.g., flip-flops, OR, AND etc. Do not use more than 4 gates!

Make a new folder under **C:\users\<login-id>\Program** and copy the folder **S:\Courses\EIT\EITF65\StopWatch**into it. Create a new project in Vivado and add all the source files from the folder **vhdl** to the project through *Add or create design sources*, refer to appendix B for instructions.

Open the source file for the **edge_det** module in Vivado and fill in the logic that implements your edge detector. The places where you should fill in code are marked with TODO.

To verify the functionality you will need to simulate your edge detector. Add all the the files from the folder **testbench** to the Vivado project through *Add or create simulation sources*. Set the simulation to execute from **edge_det_tb** module through the option *Simulation Settings->Simulation top module name* and run the simulation, for more information on this see appendix B.

During simulation the testbech will generate input signals to your edge detector, i.e different values of the signals `n_rst`, `clk` and `inp` will be generated. It is up to you to verify the functionally of the design by looking at the value/waveform of the output signal `det_edge` during different input patterns.

Show and explain the simulation result to a teaching assistant.

End of laboratory problem 4.1

Laboratory problem 4.2. Control FSM

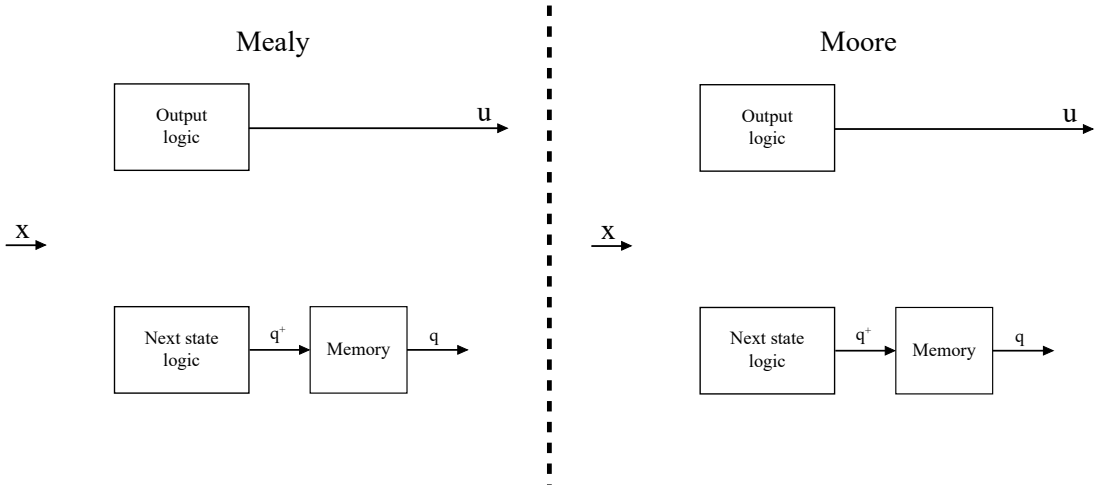
Controlling the different states of the stop watch makes the use of an FSM necessary. The FSM shall have two inputs connected to the input buttons; called `start` and `stop`. Two outputs `run_cnt` and `set0` will trigger the subsequent counter block. Output `run_cnt` is active during the run state of the stop watch and output `set0` is active for one clock cycle when resetting the counter.

Home problem 4.2.1

Draw the graph of the FSM fulfilling aforementioned specifications as Moore **and** Mealy implementation. Do you need the same number of states for both implementations?

Home problem 4.2.2

Draw arrows between the blocks to show how the blocks control each other. What is the fundamental difference between Mealy and Moore?



Open the **fsm** module in Vivado and fill in the logic that implements your FSM, implement it as either Mealy or Moore. The places where you should write code are marked with TODO. Do not forget that sequential circuits should be asynchronously reset when the input `n_rst` goes low.

Verify the functionality by simulation. For this task you need to set the simulation to start from **fsm_tb**. Show and explain the simulation results to a teaching assistant.

End of laboratory problem 4.2

The actual counting of the values is performed in the counting block; thus, it mainly consists of different counters. To be able to map counted clock cycles to actual time, knowledge about the clock frequency of the FPGA board is necessary.

Two inputs `run` and `set0` are used. While `run` is active, counter is active, otherwise the current value is held constant. The input `set0` resets the counter to zero when activated. The output of the current time is split into signals. First, `count_int` counts the seconds the stop watch has run since start and second, `count_frac` counts the hundredths of a second the stop watch has run.

Home problem 4.2.3

The Nexys-4 board is driven by a 100 MHz clock. What is the time resolution that can be achieved with this frequency, i.e., the clock period. How many clock cycles need to be counted until 1/100 s, 1/10 s and 1 s respectively, have passed? What is the number of bits required for the outputs `count_int` and `count_frac`?

Laboratory problem 4.3. Binary to NBCD decoder

The seven-segment driver has 4 inputs each generating a digit, ranging from 0 to 9, on the display. Since the outputs from the counter block are binary, ranging from 00 to 99, they need to be decoded as NBCD (Natural Binary Coded Decimal). This decoding is done in the NBCD block.

The block should split the binary input value into two NBCD outputs, i.e. if the input is $0100010_2 = 34_{10}$ the NBCD outputs should be $0011_2 = 3_{10}$ and $0100_2 = 4_{10}$.

One way to solve this is to use a look up table. But the table would be rather long as the number of rows would equal the number of input combinations, i.e. 99 in this case.

Here, another approach is used. The binary input needs to be split into tens and ones like $34 = 3 * 10 + 4$. This can be done by dividing the binary input by 10_{10}. The quotient from the division is the tens and the remainder the ones. Below 34 is divided by 10 using long division (binary values)

```
          0011
1010 | 00100010
     - 0000
       01000
     -  0000
        10001
     -   1010
        01110
     -    1010
         0100
```

the division is done by the following steps

- Shift the divisor one step to the right
- Check if the part of the dividend above the divisor is greater or equal to to 10
- If so, subtract 10 from the dividend and put a 1 in the quotient at this position

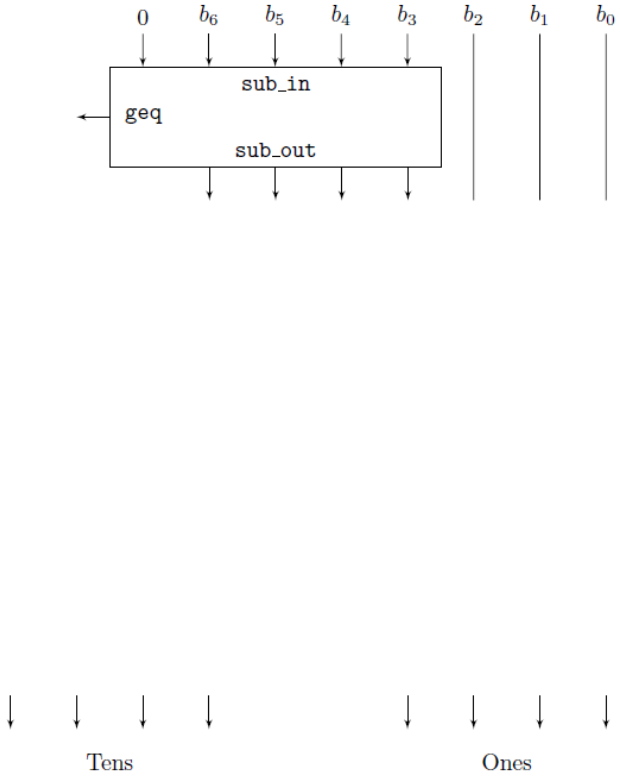
- Repeat above steps until the divisor has reached the least significant bit of the dividend

The division can be done with a combinational circuit, i.e. no clock is needed. To do this you have been provided with a combinational block called `sub10`. A 5 bit binary number is placed at the input `sub_in` of the block. If the input is greater or equal to 10_{10}, the output `sub_out` = `sub_in` $- 10_{10}$ and output `geq` $= 1$. Otherwise, `sub_out` = `sub_in` and `geq` $= 0$. Since `sub_out` is always smaller than 10, only 4 bits are needed for the output.

Compare the function of the `sub10` block with the steps in the division algorithm. Shifting of the divisor is accomplished by routing signals between `sub10` blocks.

Home problem 4.3.1

Complete the hardware implementation of the division by routing signals between `sub10` blocks. The input consists of a binary number b_6 through b_0.



Open the **nbcd** module and complete the NBCD decoder circuit. Run the simulation with **nbcd_tb**. Change the radix of the signals from binary to unsigned to ease the verification, refer to appendix B on how to do this. Show the result to a teaching assistant.

End of laboratory problem 4.3

The seven-segment driver has 4 inputs `seg0`, `seg1`, `seg2` and `seg3`, which are the NBCD values for the four seven-segment panels. The seven-segment panels share cathodes but each has a separate anode; therefore the anodes are triggered in a round-robin fashion to illuminate one number at a time. Naturally, this has to happen fast enough not to make them flicker. Moreover, this has to happen at least four times as fast as the fastest used counter, to be able to illuminate all numbers at least once before counting up a value. The driver provided in **seven_segment.vhd** performs exactly this triggering and additionally, decodes the NBCD input into a seven bit output for the seven-segment display using a LUT (look-up table).

Verify the design

When all modules are designed and the functionality of each individual module has been verified, it is time to simulate the final design.

In the **stop_watch** module all blocks are connected according to the block schematic in figure 4.3, i.e. **stop_watch** is the top level module of the design. However, for the purpose of this lab another top level module will be used during the final simulation, namely **stop_watch_sim**.

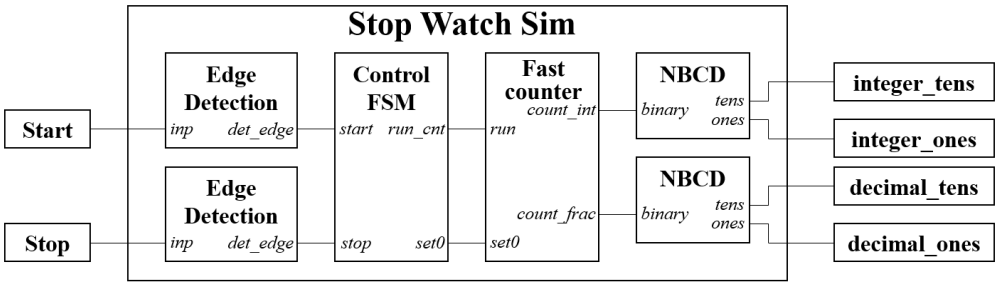


Figure 4.4: Block schematic of the design used during the final simulation.

As seen in figure 4.4 the **stop_watch_sim** module does not have any debouncers at the input. The reason for this is to cut down on time needed to run the simulation. A faster system clock will also be used to further decrease the time it takes to run the simulation. Also the seven-segment decoder has been removed to ease the verification of your design as the output of the seven-segment decoder is hard/messy to interpret.

Laboratory problem 4.4.

Configure the simulation to execute from **stop_watch_tb**. Set the simulation step time to 70 ms and run a single simulation step. This time is needed to execute all input patterns from the testbench file.

Check the outputs after different input patterns and verify that the design is behaving as specified. Show and explain the simulation to a teaching assistant.

End of laboratory problem 4.4

Prototyping on an FPGA

As a last task you will prototype the design on an FPGA. To do this the design needs to be synthesized. Then the it should be implemented with resources available on the selected FPGA. And last, the design needs be downloaded to hardware.

Before proceeding make sure to add the constraint file **top.xdc** to the project through *Add or create constraints*. The constraint file tells Vivado how signals at the top level of the design should be connected to external pins of the FPGA chip. Which in turn are connected to buttons and the seven-segment display.

Laboratory problem 4.5.

All steps needed to prototype the stop watch on an FPGA are described in appendix B. First, perform synthesis on your design. Make sure you do not have any warnings. Especially, any latches have to be removed as they can lead to unpredictable results in the design.

Have a look at your Synthesis result in Vivado. What is the hardware utilization of the FPGA?

Answer: _____

Go ahead and implement the design and then generate the bit-stream and download it to the FPGA. Make sure the Nexys4 board is powered before downloading the bit-stream.

Test your design and verify the functionality. Perform changes, if necessary, and regenerate the bit-stream. Maybe you need to run some simulations again in order to find possible malfunctions.

Bring up the RTL schematic of the design (see appendix B) and check your edge detection circuit. Is it implemented as you drew it in home problem 4.1.1?

Show the RTL schematic and demonstrate your stop watch to a teaching assistant.

End of laboratory problem 4.5

LABORATION 5

MCU

Laborationens syfte

Under denna laboration skall två uppgifter lösas. Först ska ett trafikljus i en korsning styras. Därefter ska antalet lejon i en bur räknas (likt lejonburen i laboration 1). Istället för att lösa uppgifter med två olika kretsar ska de lösas med en mer generell krets, en MCU (Micro Controller Unit).

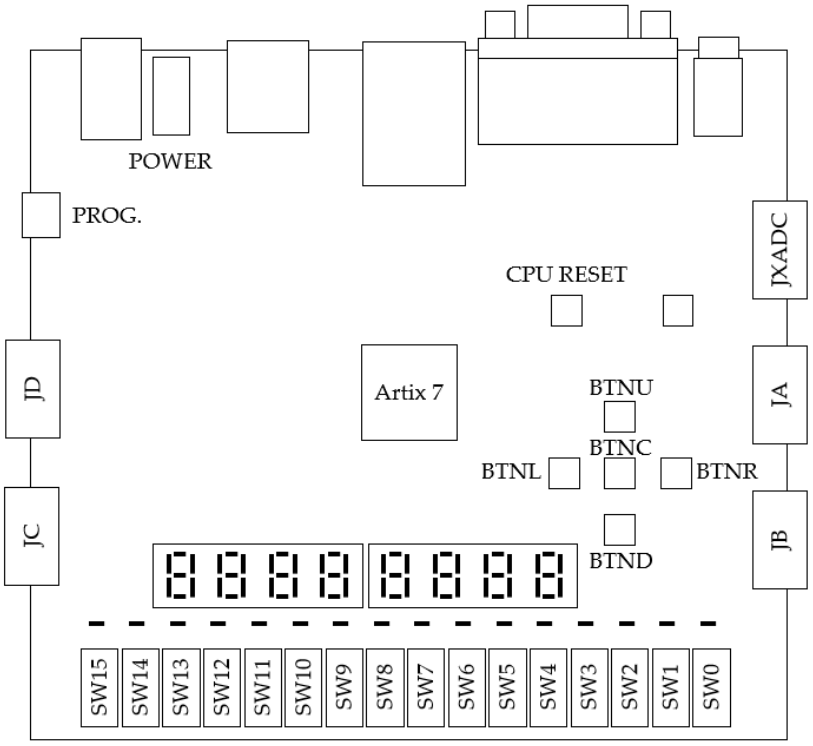
En MCU, eller enkretsdator, är en dator bestående av CPU (Central-Processing-Unit), programminne och arbetsminne allt samlat i en och samma IC-krets. En MCU kan ofta kommunicera med sin omgivning via generella in- och utsignalsportar där fysiska komponenter kan anslutas. Med en MCU kan samma krets användas för att lösa olika problem genom att modifiera det program som laddas in i dess minne.

Laborationen består av två delar. Först ska MCU:n, hårdvaran, konstrueras med hjälp av VHDL och utvecklingsverktyget Vivado från Xilinx. Hårdvaran realiseras på utvecklingskortet Nexys4. Därefter ska två program som löser de två uppgifterna skrivas.

Följande färdigskrivna VHDL-komponenter finns tillgängliga under laborationen:

Komponent/Modul	Filnamn	Beskrivning
`SR4`	**sr4.vhd**	4-bitars skiftregister
`REG6`	**reg6.vhd**	6-bitars register
`MUX3x6`	**mux3x6.vhd**	6-bitars 3- till 1-multiplexer
`ALU8`	**alu8.vhd**	8-bitars ALU
`REG8`	**reg8.vhd**	8-bitars register
`MUX2x8`	**mux2x8.vhd**	8-bitars 2- till 1-multiplexer

Nexys4 FPGA utvecklingskort



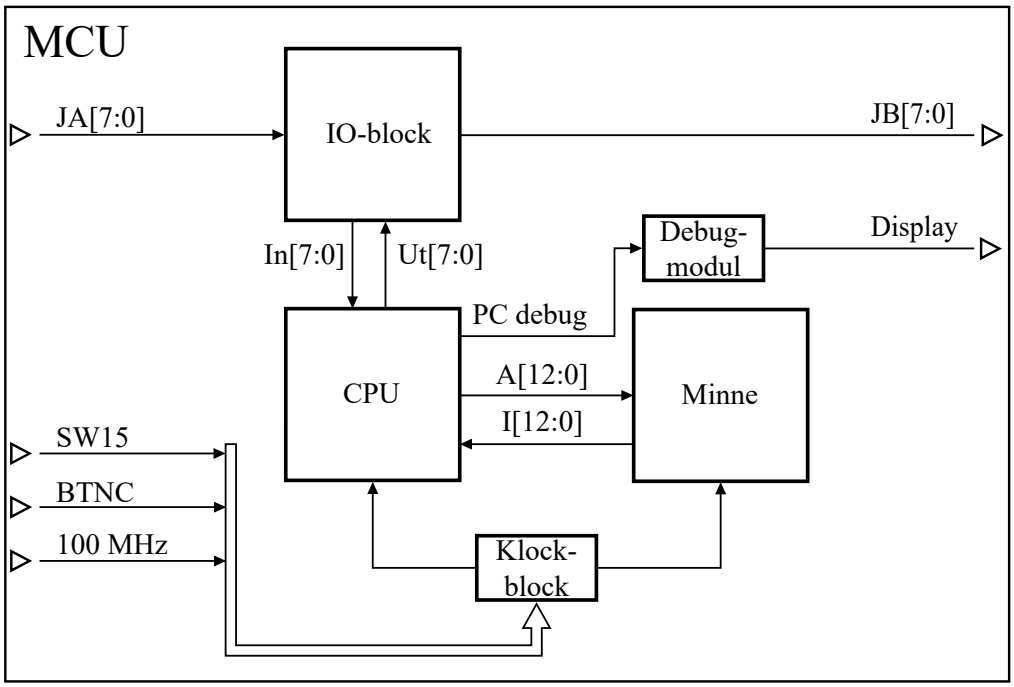
Figur 5.1: Strömställarna `SW0` till `SW15` genererar en logisk etta då de är i sitt övre läge. Knapparna som är arrangerade i ett kors, `BTNx`, genererar en logiskt etta då de trycks ner. Knappen `CPU RESET` fungerar på motsatt sätt, nolla genereras då den trycks ner. Anslutningarna `JA–JD` är generella in- och utsignalsportar som kan kopplas till yttre komponenter.

Introduktion

Läs igenom hela manualen innan du börjar beskriva hårdvaran i VHDL!

Under laboration ska hårdvaran för en MCU (Micro Controller Unit) konstrueras med hjälp av VHDL. Därefter ska konstruktionen realiseras på ett Nexys4 FPGA-kort. Den färdiga MCU:n kommer att kunna utföra ett antal olika operationer, exempelvis addera två tal och spara resultatet. MCU:n styrs genom att ett program laddas ner i dess minne. Programmet kommer att bestå av en lista med instruktioner som talar om för MCU:n vilka operationer den ska utföra och i vilken ordning.

Efter att hårdvaran har implementerats ska två olika program skrivas och laddas ner i MCU:n för att lösa de två tidigare nämda uppgifterna (trafikljus vid en vägkorsning och lejonburen).



Figur 5.2: Blockschema av MCU:n som ska konstrueras. Observera att resetsignalen inte är utritad.

Hårdvarudelen av laborationen består av att konstruera CPUn (Central Processing Unit) då övriga block i figur 5.2 tillhandahållas som färdiga VHDL moduler.

MCU:ns minne är av typen ROM (Read Only Memory). Därför kan kretsen inte modifiera dess innehåll utan endast läsa av det. Programmet som styr MCU:n kommer att programmeras in i minnet så att varje instruktion i programmet sparas på en given rad, minnesadress. Programmeringen av minnet görs med ett PC-program som beskrivs senare.

När ett program körs kommer debug-blocket att skriva ut minnesadressen till den instruktionen som exekveras för stunden, på 7-segmentsdisplayen. Detta kan vara använd-

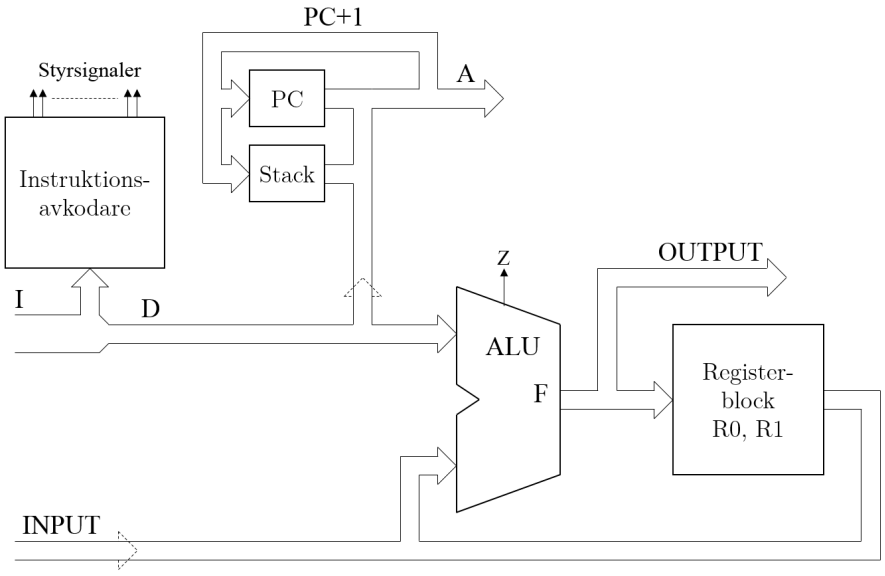
bart vid felsökning av program.

Hela konstruktionen drivs av en 1 kHz-klocka som genereras i klockblocket utifrån 100 MHz-klockan på Nexys4. Det finns även en manuell klocka `BTNC` som kan användas för att stega genom programmen. Valet mellan 1 kHz och manuell klocka görs med hjälp av brytaren `SW15`, se figur B.1. Varje instruktion i programmen kommer att ta en klockcykel för MCU:n att utföra.

Externa komponenter som lysdioder och brytare kommer även att anslutas till MCU:n under laborationen. Komponenter som genererar insignaler till kretsen ansluts till kontakten `JA` och komponenter som ska visa utsignaler från kretsen ansluts till `JB`. Både `JA` och `JB` består av åtta bitar och synkroniseras med resten av kretsen i IO-blocket (Input-Output). De synkroniserade versionerna av `JA` och `JB` är signalerna `INPUT` respektive `OUTPUT` i figur 5.3.

På 7-segmentsdisplayen åskådliggörs minnesadressen till den instruktionen som exekveras för stunden. Detta sker med hjälp av debug-blocket. Denna visuella återkoppling i kombination med den manuella klockan är ett ypperligt verktyg vid felsökning av ett program. Man kan således ta sig igenom sitt program steg för steg och samtidigt se vilken instruktion som exekveras.

Då ett program exekveras läser CPU:n av en instruktion `I` från en adress `A` i minnet, se figur 5.3. CPU:n avkodar instruktionen, utför nödvändiga operationer och genererar adressen till nästa instruktion som ska läsas från minnet.



Figur 5.3: Abstrakt blockschema av CPU:n som ska konstrueras.

För att ett program ska exekveras korrekt måste CPU:n på något sätt komma ihåg var i programmet den befinner sig. CPU:n ska därför ha en programräknare `PC` (Program-Counter), som är ett register[5] vars värde alltid är adressen till den instruktion som körs

[5]Ett register består av ett antal parallellkopplad D-vippor. Register används då man behöver ett minne

för stunden, se figur 5.3. Stacken i figuren kommer att förklaras senare men kan ses som ett minne där programadresser kan sparas för att vid ett senare tillfälle läsas.

De beräkningar som utförs i CPU:n kommer att göras med en ALU. Den ena operanden till ALU:n kommer alltid att komma från en instruktion som hämtas från minnet. Den andra operanden kommer antingen att vara `R0`, `R1` eller signalen `INPUT`. Värdet av utsignalen `F` från ALU:n sparas alltid i ett av de två registerna `R0` och `R1` i registerblocket. Konstruktionen blir därmed begränsad i det avseendet att det inte går att utföra några operationer mellan de båda registerna `R0` och `R1`.

Instruktionsavkodaren i CPU:n har till uppgift att utifrån instruktionen `I` generera styrsignaler till alla komponenter i CPU:n så att rätt operation erhålls.

Instruktioner

De instruktioner som en CPU stödjer kallas dess instruktionsuppsättning och utgör gränssnittet mellan hård- och mjukvara. Instruktionsuppsättningen specificerar även kraven för hårdvaran då kretsen måste kunna utföra de olika instruktionerna. CPU:n som konstrueras under laborationen kommer endast stödja ett fåtal instruktioner men kan enkelt byggas ut för att stödja fler.

Varje instruktion består av en 4-bitars operationskod, en destinationsbit samt 8-bitars data, totalt 13 bitar. Med operationskoden `OPCODE` bestäms vilken operation som ska utföras. Destinationsbiten `DEST` styr i vissa instruktioner vilket av de två registerna `R0` och `R1` som ska användas. De 8 databitarna `DATA` används som operand i beräkningar eller för att ange adressen till nästa instruktion. Bitarna i varje instruktion organiseras enligt figur 5.4.

Figur 5.4: Varje instruktion, `I`, består av 13 bitar. Bit 12-9 är instruktionens operationskod, bit 8 destinationsbiten och bit 7-0 är data. Observera att bit 0 är den minst signifikanta.

$$\texttt{RX} = \begin{cases} \texttt{R0} & , \texttt{DEST} = 0, \\ \texttt{R1} & , \texttt{DEST} = 1. \end{cases}$$

som kan spara mer än en bit. T.ex. består ett 8-bitars register av åtta D-vippor och har 2^8 olika tillstånd.

Nedan följer en beskrivning av de instruktioner som CPU:n ska stödja. Istället för 4-bitars OPCODE anges instruktionerna med förkortningar för att de ska vara lättare att komma ihåg. I listan nedan gäller att

CALL Anropa subrutin (CALL subroutine). Nästa adress i ordningen PC+1 sparas (PUSH) på stacken och nästa adress bestäms av DATA[5:0]. Kan användas då man har en programsnutt som ska köras från flera ställen i programmet, liknande ett funktionsanrop i t.ex. Java och C.

RET Återgå från subrutin (RETurn from subroutine). Nästa adress hämtas och tas bort (POP) från stacken. Används endast i slutet av en subrutin för att återgå till huvudprogrammet.

BZ Hoppa om noll (Brach if Zero). Om RX=0 så bestäms nästa adress av DATA[5:0], annars är nästa adress PC+1. Används för att hoppa över/till programkod beroende på om ett villkor är uppfyllt, liknande if-satser i Java och C.

B Hoppa (Branch). Ovillkorligt programhopp till adressen som anges av DATA[5:0]. Kan användas i slutet av programmet för att börja om från adress 0.

ADD Addera (ADD). Värdet DATA adderas till RX.
RX^{+}=RX+DATA.

SUB Subtrahera (SUBtract). Värdet DATA subtraheras från RX.
RX^{+}=RX−DATA.

AND Bitvis OCH (AND). Bitvis OCH-funktion mellan RX och DATA.
RX[i]$^{+}$=RX[i]∧DATA[i], i=0..7.

LD Ladda (LoaD). Ladda RX med DATA.
RX^{+}=DATA.

OUT Skriv till utsignal (write to OUTput). Utsignalen får värdet som finns i register RX.
OUTPUT^{+}=RX.

IN Läs från insignal (read from INput). RX laddas med värdet av insignalen.
RX^{+}=INPUT.

Med instruktionerna ovan kan ett enkelt program skrivas som

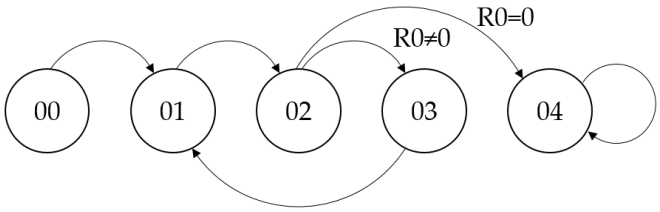
```
ADRESS  OPCODE  DEST  DATA
00      LD      R0    2
01      SUB     R0    1
02      BZ      R0    04
03      B             01
04      B             04
```

På progamadress 00 laddas R0 med värdet 2, därefter minskas R0 med ett. På adress 02 kontrolleras om R0=0. Om så är fallet hoppar programmet till 04, annars fortsätter det

på 03 där det hoppar tillbaka till 01. När programmet till slut kommer fram till adress 04 kommer det stå kvar där tills kretsen nollställs.

$$00 \rightarrow 01 \rightarrow 02 \rightarrow 03 \rightarrow 01 \rightarrow 02 \rightarrow 04 \rightarrow 04 \ldots$$

Programflödet kan även visualiseras med ett tillståndsdiagam där tillståndskoderna motsvarar adresserna i programmet, se figur 5.5.



Figur 5.5: Tillståndsdiagrammet visualiserar i vilken ordning programadresserna exekveras, en form av programflödesdiagram.

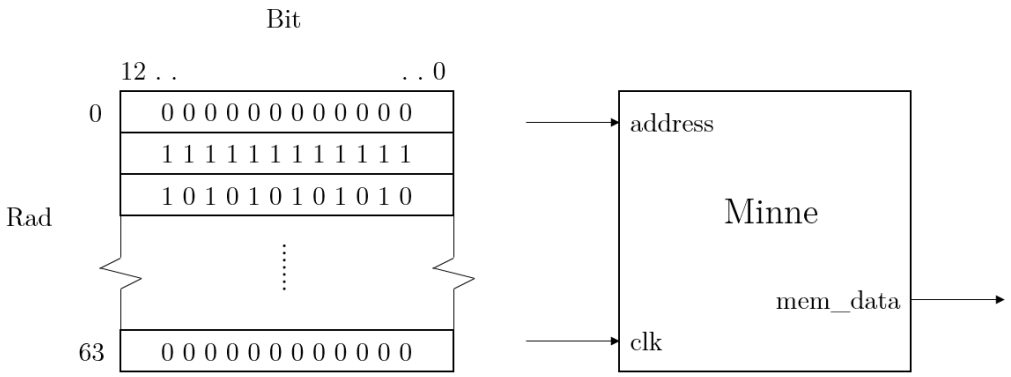
Då minnet som används under laborationen kommer ha begränsad kapacitet bör instruktionsstycken återanvändas om så är möjligt. I programmet nedan används instruktionerna på adress 10 till 14 som en subrutin för att generera en fördröjning.

```
ADRESS  OPCODE  DEST  DATA
00      IN      R0
01      OUT     R0
02      BZ      R0    06
03      SUB     R0    1
04      CALL          10
05      B             01
06      CALL          10
07      B             00
...
10      LD      R1    2
11      SUB     R1    1
12      BZ      R1    14
13      B             11
14      RET
```

Programmet börjar med att ladda `R0` med värdet av insignalen `INPUT`. Värdet av `R0` skrivs sedan till utsignalen `OUTPUT`. Om `R0`=0 hoppar programmet till 06 annars minskas `R0` med 1 och subrutinen anropas vid 04. När subrutinen är slut, på adress 14, återgår programmet till adress 05, alltså raden efter subrutinsanropet. Därefter upprepas proceduren tills R0 är 0 då programmet hoppar 06 och subrutinen anropas än en gång. Efter det körs programmet ifrån början igen.

Minne

Minnets storlek bestäms av två parametrar, antal rader och antal bitar per rad. Under laborationen används ett minne med 64 rader där varje rad består av 13 bitar, se bild 5.6.

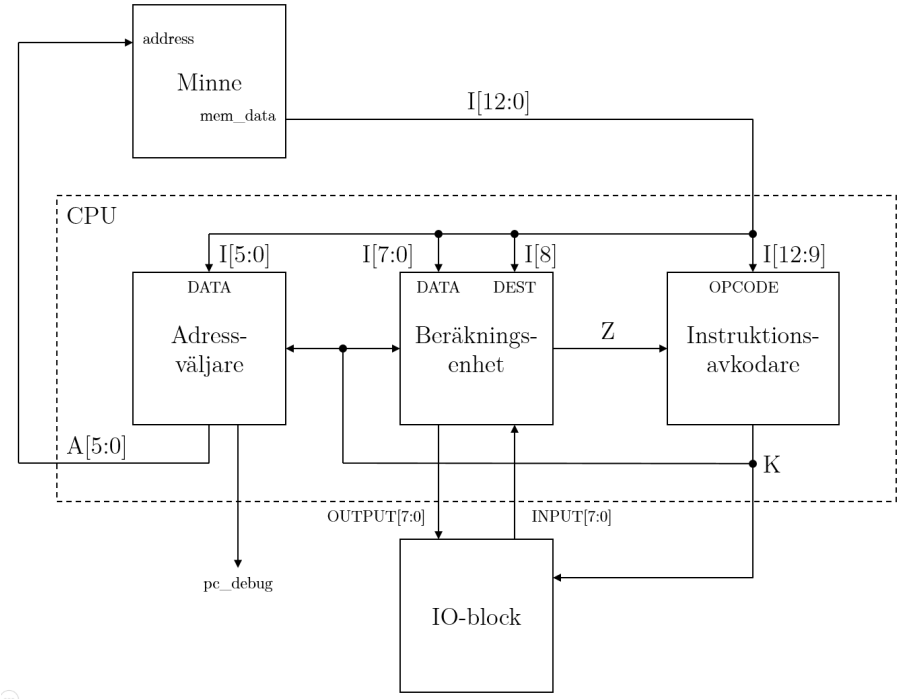


Figur 5.6: 64x13bit minne.

Med insignalen `address` bestäms vilken rad som ska finnas på utsignalen `mem_data` vid nästa stigande flank hos signalen `clk`. För att nå alla 64 rader måste signalen `address` bestå av 6 bitar och `mem_data` måste vara 13 bitar bred för att hela raden ska kunna läsas ut. Minnet kommer att utnyttjas så att varje programrad lagras på varsin minnesrad.

Del 1. Hårdvara - Konstruktion av CPU

I denna del av laborationen ska hårdvaran för CPU:n konstrueras. Konstruktionen kommer att delas upp i tre delar; adressväljare, beräkningsenhet och instruktionsavkodare enligt figur 5.7. Först ges en kort beskrivning av de olika blocken. Därefter ska de olika blocken konstueras och testas för att slutligen sättas ihop till en fungerande CPU.



Figur 5.7: Blockschema för CPU:n. För att göra schemat tydligare symboliserar signalen `K` följande sex signaler: `StackOp[1:0]`, `AddrSrc[1:0]`, `ALUOp[2:0]`, `ALUSrc`, `OutEna` och `RegEna`.

Beräkningsenheten utför aritmetiska och logiska operationer samt läser och skriver till IO-blocket. Signalen `Z` används till att styra villkorliga instruktioner. När resultatet av en operation i ALU:n är noll ska utsignalen (flaggan) `Z` bli aktiv (hög).

Instruktionsavkodaren avkodar instruktionen `I` från minnet. Beroende på instruktion och signalen `Z` genereras styrsignaler till alla delar av CPU:n samt till IO-blocket.

I adressväljaren genereras adressen till nästa instruktion. Adressväljaren ska även innehålla programräknaren `PC` vars värde alltid ska vara adressen till instruktionen som exekveras för stunden.

5.1. Adressväljaren

Beroende på instruktionen som exekveras kommer nästa adress att genereras på olika sätt. Det är adressväljarens uppgift att utifrån styrsignaler från instruktionsavkodaren generera rätt adress till minnet.

Vid subrutinsanrop med instruktionen `CALL` så måste återhoppsadressen sparas undan.

Detta eftersom CPU:n måste veta vilken programrad den ska hoppa till då subrutinen är slut, vid instruktionen `RET`.

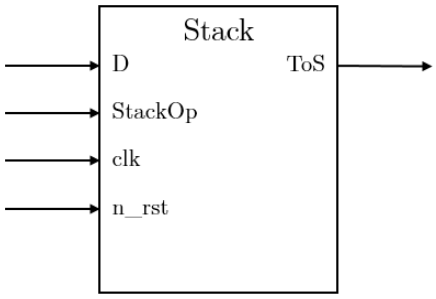
För att spara återhoppsadressen ska en stack användas. Stacken arbetar enligt principen att det som senast sparats där är det som först läses ut, det vill säga ett minne utav LIFO-typ (Last In First Out).

I detta avsnitt ska först en stack realiseras. Denna tillsammans med andra komponenter ska sedan användas för att realisera adressväljaren.

Uppgift 5.1.1. Implementering av stacken

Stacken som ska implementeras ska bestå utav fyra minnesceller. I varje cell ska en adress kunna lagras, alltså måste varje cell kunna hålla 6 bitar. Med 4 minnesceller kan lika många successiva subrutinsanrop göras. Skulle ytterligare ett göras kommer återhoppsadressen från den första subrutinen att gå förlorad och risken är stor att programmet inte beter sig korrekt.

I figur 5.8 ses stacken med dess in- och utsignaler, observera antalet bitar för de olika signalerna. Stackens tillstånd ändras vid klocksignalens `clk` stigande flank. Utsignal `ToS` (Top-of-Stack) ges av stackens översta cell. Signalen `n_rst` är en synkront aktiv låg resetsignal som nollställer stackens celler. Signal `StackOp` ska styra stacken enligt tabellen nedan



Figur 5.8: Symbol av stacken som ska implementeras. Anslutningarna D och ToS är 6 bitar breda och StackOp 2 bitar bred.

`StackOp`	Funktion
PUSH	Värdet av D sparas överst på stacken. Tidigare sparade värdet trycks ned ett steg.
POP	Stackens innehåll flyttas upp ett steg och den understa cellen får värdet 0.
HOLD	Stackens innehåll oförändrat

Nedan ges ett exempel på hur stacken ska bete sig vid de olika operationerna.

`StackOp` = PUSH
`D` = 20

Före:

`ToS`	2
	0
	0
	0

Efter:

`ToS`	20
	2
	0
	0

`StackOp` = POP

Före:

`ToS`	1
	2
	3
	4

Efter:

`ToS`	2
	3
	4
	0

`StackOp` = HOLD

Före:

`ToS`	1
	2
	3
	0

Efter:

`ToS`	1
	2
	3
	0

För att implementera stacken ska sex skiftregister `SR4` användas. Dessa kopplas parallellt så att varje skiftregister kan spara bitar med en given position från alla 4 minnesceller. Till exempel ska ett skiftregister spara bit 0 från alla celler, ett annat sparar bit 1 från alla celler och så vidare. Operationerna PUSH och POP blir då det samma som att höger- respektive vänsterskifta innehållet i alla register.

Hemuppgift 5.1.1.1

Använd 6 skiftregister SR4 och anslut kretsarna för att realisera stacken enligt beskrivningen ovan. Rita nätet samt sätt ut signalerna från figur 5.8. Beskrivning av SR4 finns i appendix C.

Fyll värdet för signalen `StackOp` vid de olika operationerna.

Operation	`StackOp[1:0]`
POP	
PUSH	
HOLD	

Alla filer som behövs till laborationen finns i katalogen **S:\Courses\EIT\EITF65\MCU**. Kopiera hela mappen till **C:\users\<login-id>\Program**. Mappen innehåller fem undermappar. I **komponenter** finns färdiga moduler som kan användas, dessa finns även listade i början av manualen. Innehållet i **misc** ska användas vid syntetiseringen och implementeras i Vivado. Mappen **tb** innehåller alla simuleringsfiler. I **labbfiler** finns VHDL-filer som beskriver MCU:n och i mappen **fpga_rom_editor** finns PC-programmet som används för att skriva och ladda ner program till MCU:n.

Börja med att skapa ett nytt projekt i Vivado, en beskrivning av hur man gör finns i appendix B. Lägg till alla källfiler från mapparna **komponenter**, **labbfiler** och **misc** i projektet genom *Add or create design sources*. Lägg även till alla simuleringsfiler från mappen **tb** via *Add or create simulation source*.

För att undvika fel som beror på att signalvärden tolkas på olika sätt i olika filer används ett *package* där konstanter definieras. Dessa kan sedan användas i andra filer.

Öppna filen **cpu_pkg.vhd** och fyll i konstanterna för signalen `StackOp` som styr stackens operation enligt hemuppgiften ovan. Eftersom **cpu_pkg.vhd** inte är en VHDL-modul finns den inte listad under *Sources->Hierarchy* i Vivado. Filen hittas istället genom att man

väljer fliken *Sources->Libraries*. Glöm inte att avkommentera raderna där konstanterna är deklarerade.

Öppna därefter modulen **stack** och implementera stacken genom att instansiera och koppla ihop 6 skiftregister SR4 (definierade i filen **sr4.vhd**). Använd konstanterna från **cpu_pkg** för att styra stackens funktion.

Om en utsignal från en komponent inte behövs kan denna anslutas till *open*. Ordet *open* är reserverat i VHDL och talar om för Vivado att utsignalen inte används och kan tas bort vi optimering, se exempel nedan.

```
instancename:  entity adder port map(
     adder_input => signalA,
     adder_out1 => signalB,
     adder_out2 => open
);
```

Verifiera stackens beteende genom att köra simuleringen **stack_tb**. Det kan underlätta att öppna **stack_tb** och läsa igenom kommentarerna för att förstå vad som händer.

Under simuleringar finns signalen `message` tillgänglig i vågformsfönstret. Denna är till för att identifiera vilken rad i testfilen som genererar insignaler till modulen.

Kontrollera så att stacken beter sig korrekt.

Slut på uppgift 5.1.1

Uppgift 5.1.2. Implementering av adressväljaren

Stacken ska användas för att konstruera adressväljaren som ska generera adressen `A` till nästa instruktion i minnet.

Antag att instruktionen som exekveras finns på platsen `PC` i minnet. Beroende på instruktionstyp ska adressen `A` till nästa instruktion genereras enligt nedan

- Vid instruktioner som inte påverkar programflödet, som `ADD` och `OUT`, ges nästa adress av `A = PC+1`.
- Vid hoppsinstruktionen `B` ges nästa adress av de 6 minst signifikanta bitarna ur instruktionskoden, dvs `A=DATA[5:0]`.
- Vid villkorligt hopp `BZ` ges nästa adress av `A=DATA[5:0]` om resultatet `F` från ALU är noll. Annars ges nästa adress av `A = PC+1`.
- Vid subrutinsanrop med instruktionen `CALL` gäller att `A=DATA[5:0]`. Dock så måste återhoppsadressen `PC+1` sparas (PUSH) på stacken.
- Vid instruktionen `RET` ges nästa adress av det översta elementet (POP) på stacken `A=ToS`.

För att kunna utföra ovanstående måste adressväljaren

- Innehålla ett minne/register som hela tiden sparar adressen till instruktionen som exekveras, dvs en programräknare `PC`. Vid varje stigande flank hos klockan ska nästa adress `A` sparas i programräknaren, $\texttt{PC}^{+} = \texttt{A}$.

- Kunna addera ett till programräknaren för att erhålla `PC+1`.

- Innehålla en stack där `PC+1` kan sparas på toppen

- Innehålla en komponent som beroende på värdet av `AddrSrc[1:0]` sätter nästa adress `A` till ett av följande värden

Nästa adress `A`	**Beskrivning**
`PC+1`	Nästa adress i ordningen.
`ToS`	Det översta värdet på stacken (Top-of-Stack).
`DATA[5:0]`	De 6 minst signifikanta bitarna i instruktionen.

Insignaler till adressväljaren är `clk`, `n_rst`, `DATA[5:0]`, `StackOp[1:0]` och `AddrSrc[1:0]`. Utsignaler är `A` och `pc_debug`.

Utsignalen `pc_debug` ska få sitt värde från programräknaren, `pc_debug` = `PC`. Detta för att debug-modulen ska fungera senare när program körs

Hemuppgift 5.1.2.1

Rita schemat för adressväljaren enligt tidigare specifikation och sätt ut in- och utsignaler. Tillgängliga komponenter finns listade i appendix.

Fyll i värdet för signalen `AddrSrc`

`A`	`AddrSrc[1:0]`
`PC+1`	
`ToS`	
`DATA[5:0]`	

Implementera schemat från hemuppgiften i modulen **adressvaljare**. Fyll även i konstanterna för `AddrSrc` i **cpu_pkg** så att de stämmer överens med implementationen. Observera att en utsignal deklarerad med *out* inte kan läsas av i en modul. Använd istället en intern signal vars värde sedan tilldelas utsignalen.

Om en bred signal ska få sitt värde av en smalare kan syntaxen nedan användas. I exemplet kopplas de 4 mest signifikanta bitarna av signalen `EIGHT` till konstant 0 (`EIGHT` är 8 bitar bred och `FOUR` är 4 bitar bred).

```
EIGHT <= "0000" & FOUR;  -- De fyra översta bitarna fylls med nollor.
```

Omvänt om en smal signal ska få sitt värde av en bredare kan syntaxen nedan användas. I exemplet får signalen `FOUR` sitt värde av de fyra minst signifikanta bitarna av signalen `EIGHT`

```
FOUR <= EIGHT(3 downto 0);  -- Endast de fyra lägsta bitarna används.
```

Verifiera implementationen genom att köra simuleringen **adressvaljare_tb**. Då signalen `A` ändras via ett kombinatoriskt nät kan det vara svårt att verifiera om blocket beter sig korrekt. Titta istället på utsignalen `pc_debug` som uppdateras vid varje klockflank och har värdet `pc_debug = PC`.

Slut på uppgift 5.1.2

5.2. Beräkningsenhet

Det andra blocket av CPU:n, beräkningsenheten, ska nu konstrueras. Beräkningsenheten ska förutom ALU:n innehålla registerblocket vilket är det enda minnet i MCU:n där temporär data kan sparas.

Först ska registerblocket innehållande de två 8-bitars registerna `R0` och `R1` konstrueras. Därefter ska registerblocket användas till att implementera beräkningsenheten.

Uppgift 5.2.1. Implementering av registerblocket

Registerblocket styrs med fem signaler, `RegEna`, `DEST`, `F`, `clk` och `n_rst`, och har en utsignal `RegOut`.

Signalen `DEST`, bit 8 från instruktionen, bestämmer vilket register som är aktivt och finns på utgången `RegOut` enligt

$$\texttt{RegOut} = \begin{cases} \texttt{R0}, & \texttt{DEST} = 0, \\ \texttt{R1}, & \texttt{DEST} = 1. \end{cases}$$

Om `RegEna=1` ska det aktiva registret uppdateras med värdet av insignalen `F` vid stigande klockflank, annars ska det aktiva registret vara oförändrat. Signalen `n_rst` nollställer alla register synkront då den är låg.

Hemuppgift 5.2.1.1

Rita nätet för registerblocket innehållande `R0` och `R1` med tillgängliga komponenter och vanliga grindar, t. ex. AND, NOT osv. Ange in- och utsignaler.

Implementera logiken från hemuppgiftem i modulen **registerblock**. Logiska funktioner mellan signaler som AND och NOT implementeras med VHDL syntax. Verifiera registerblocket genom att köra simuleringen **registerblock_tb**.

Slut på uppgift 5.2.1

Uppgift 5.2.2. Implementering av beräkningsenhet

Beräkningsenheten har följande insignaler `DEST`, `ALUOp`, `RegEna`, `INPUT`, `ALUSrc`, `DATA`, `clk` och `n_rst`.

Utsignaler från enheten är `OUTPUT` och `Z`. Båda genereras från en ALU som tar två 8-bitars operander, `A` och `B`, och utför operationen som bestäms av signalen `ALUOp`. Resultatet `F` från ALU:n ska anslutas till registerblocket och beräkningsenhetens utsignal `OUTPUT`. ALU:n har även en statussignal `Z` som ska anslutas till utsignalen med samma namn.

Operand `A` ska alltid vara de åtta minst signifikanta bitarna ur instruktionen `DATA[7:0]`. Operand `B` ska antingen vara `RegOut` från registerblocket eller signalen `INPUT`. Valet av `B` gör med `ALUSrc` enligt

$$\texttt{B} = \begin{cases} \texttt{RegOut}, & \texttt{ALUSrc} = 0, \\ \texttt{INPUT}, & \texttt{ALUSrc} = 1. \end{cases}$$

Signalen `Z` från ALU:n som är hög då `F=0` ska senare användas till att styra villkorliga instruktioner i instruktionsavkodaren. `OUTPUT` kommer att anslutas till kontakten `JB` på Nexys4 via IO-blocket.

Hemuppgift 5.2.2.1

Rita nätet för beräkningsenheten bestående av registerblock, ALU samt övriga komponenter som krävs. Insignal är `DEST`, `ALUOp`, `RegEna`, `INPUT`, `ALUSrc`, `DATA`, `clk` och `n_rst`. Utsignaler är `OUTPUT` och `Z`.

Implementera beräkningsenheten i modulen **berakningsenhet** och testa dess funktion med simuleringsfilen **berakningsenhet_tb**.

Slut på uppgift 5.2.2

5.3. Instruktionsavkodare

Det sista blocket av CPU:n, instruktionsavkodaren, ska nu konstrueras så att rätt styrsig-

naler genereras till de övriga blocken vid olika instruktionerna.

Uppgift 5.3.1. Implementering av instruktionsavkodare

Instruktionsavkodaren ska utifrån fältet `OPCODE` från instruktionen `I` generera rätt styrsignaler till adressväljaren, beräkningsenheten och IO-blocket. Signalen `Z` från beräkningsenheten ska användas för att styra villkorliga instruktioner.

Utsignaler från instruktionsavkodaren är de som använts i tidigare uppgifter `AddrSrc`, `StackOp`, `ALUOp`, `ALUSrc`, `RegEna` samt signalen `OutEna`.

`OutEna` ska styra när IO-blockets utsignalbuffert ska laddas. Ett högt värde hos `OutEna` medför att utsignalen `JB` tilldelas värdet av `OUTPUT` från beräkningsenheten vid nästa klockflank.

Hemuppgift 5.3.1

Fyll i tabellen nedan så att instruktionerna implementeras enligt specifikationen tidigare i manualen. Koda även varje instruktion med fyra bitar, `I[12:9]`.

Observera att det finns utrymme för att lägga till egna instruktioner om man vill. T.ex. kan en instruktion där `DATA` skrivs direkt till `OUTPUT` vara användbar senare i laborationen.

OPCODE	I[12:9]	Z	StackOp	AddrSrc	ALUOp	ALUSrc	OutEna	RegEna
CALL		-						
RET		-						
BZ		0						
BZ	`----`	1						
B		-						
ADD		-						
SUB		-						
LD		-						
IN		-						
OUT		-						
AND		-						

`StackOp` $\in$ { PUSH, POP, HOLD }
`AddrSrc` $\in$ { PC+1, ToS, DATA[5:0] }
`ALUOp` $\in$ { A, B, A+B, B-A, A $\wedge$ B, A $\vee$ B, A $\oplus$ B , 0}
`ALUSrc` $\in$ { RegOut, Input }

Än så länge har de olika instruktionerna bara angetts med förkortningar. I kretsen måste dessa dock kodas binärt med fyra bitar. Öppna **cpu_pkg** och definera bitmönster för de olika instruktionerna enligt tabellen i hemuppgiften. Det spelar ingen roll hur dessa kodas så länge man är konsekvent och använder samma kodning då programmen skrivs.

Öppna modulen **instruktionsavkodare** och implementera avkodaren enligt tabellen ovan. Se till att använda konstanterna från **cpu_pkg** för att avkoda signalen `OPCODE`.

Testa därefter instruktionsavkodaren genom att köra **instruktionsavkodare_tb**. Kontrollera att utsignalerna stämmer överens med tabellen i hemuppgiften.

Slut på uppgift 5.3.1

5.4. CPU

Som sista uppgift av hårdvarudelen ska de tre blocken adressväljare, beräkningsenhet och instruktionsavkodare kopplas ihop för att realisera CPU:n. Därefter ska MCU:n syntetiseras och implementeras i Vivado.

Uppgift 5.4.1. Implementering av CPU

Öppna modulen **cpu** och instansiera och koppla ihop de olika blocken enligt figur 5.7. Tänk på att ersätta signalen `K` i figuren med de verkliga signalerna.

Slutligen ska hela CPU:n simuleras med filen **cpu_tb**. Under simuleringen kommer ett program exekveras genom att olika instruktioner `I` genereras till CPU:n. Simuleringsfilen kommer efter varje instruktion kontrollera så att rätt utsignaler generats från CPU:n. Om ett fel skulle inträffa kommer detta att skrivas ut i meddeladefönstret i Vivado. Programmet som simuleras finns listat i början av simuleringsfilen.

Observera att simuleringen inte kontrollerar alla funktioner hos CPU:n. Så även om inga fel hittas finns det inga garanti för att den är felfri.

Om ett fel skulle inträffa kan det vara lämpligt att köra igenom de tidigare simuleringarna för att hitta var felet är. Du kan använda dig av RTL-schemat för att hitta eventuella felkopplingar.

Uppgift 5.4.2. Syntetisering och implementering av MCU

Se till att du lagt till filen **Nexys4_Master.xdc** i projektet. Syntetisera, implementera och generera bitströmmen för projektet i Vivado.

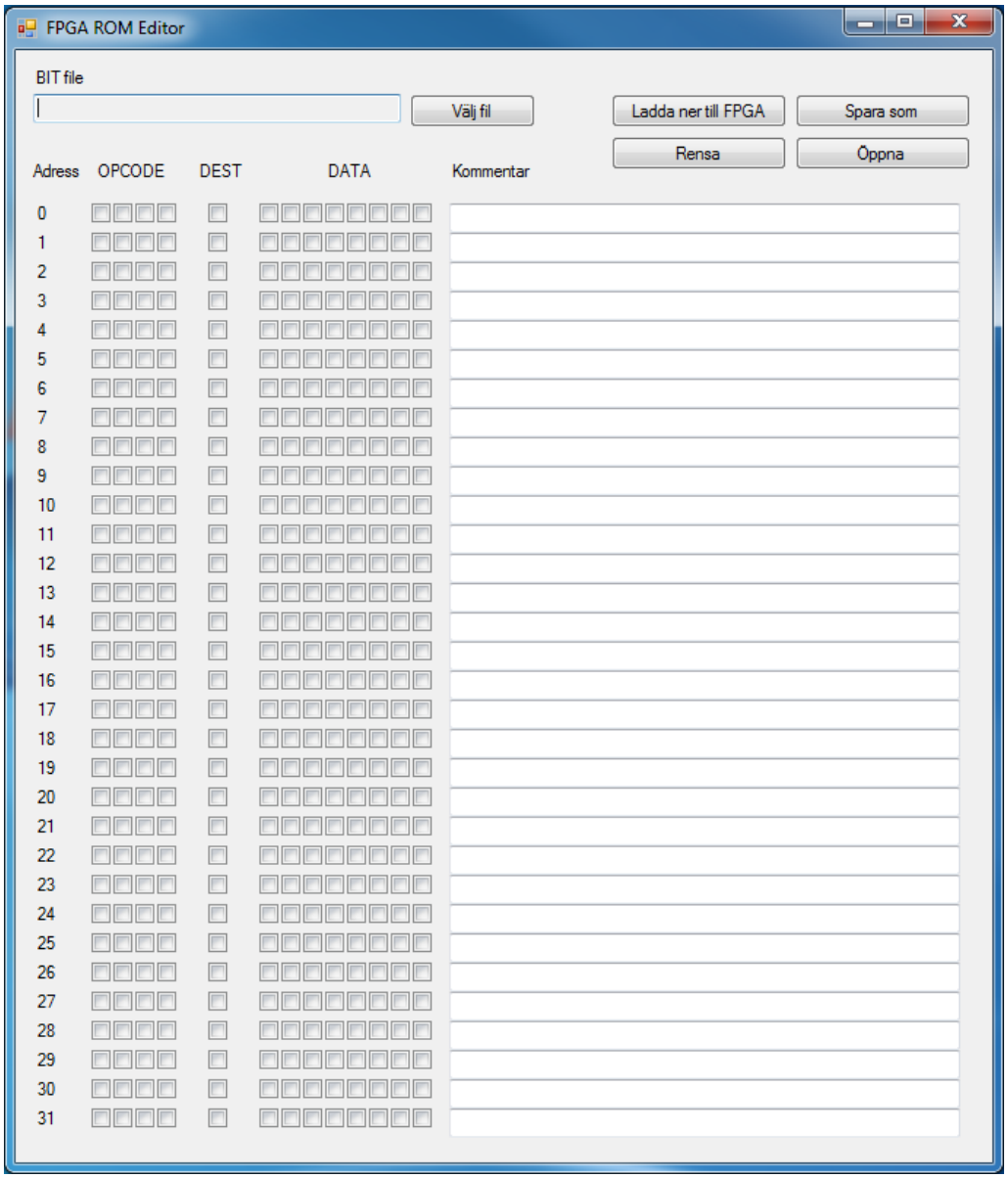
Slut på uppgift 5.4.2

Del 2. Mjukvara - Programmering av MCU

Med en väl fungerande MCU återstår nu bara att skriva programmen som löser de två problemen trafikkorsningen och lejonburen.

Minnesmodulen i MCU:n är avsedd att innehålla programmet som styr enhetens bete-

ende. Efter att bitströmmen generats i Vivado fylls hela minnesarean med nollor. Bitströmmen måste därför modifieras så att ditt program placeras i minnet. Modifieringen av bitströmmen görs med programmet **FPGA_ROM_Editor.exe** som finns i mappen **rom_editor**.



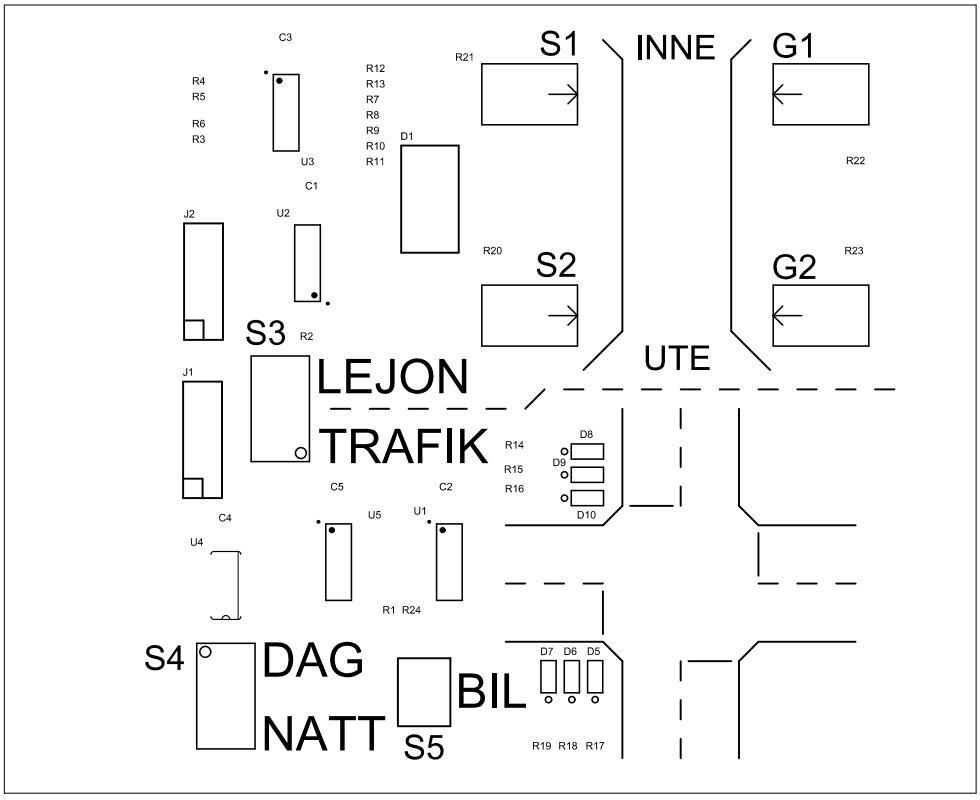
Figur 5.9: Skärmklipp av FPGA_ROM_Editors gränssnitt. Programmet används för att kopiera din kod till MCU:ns minne innan designen laddas ner till FPGA-kortet.

Figur 5.9 visar hur programmet ser ut då det startats. Antalet programrader (instruktioner) är 64 och varje instruktion består av en 4-bitars operationskod, en destinationsbit och 8 bitar data (MSB ... LSB). En bock i en ruta markerar att motsvarande bit är satt till ett, annars noll. Efter varje instruktionsrad finns det en ruta där du kan skriva kommentarer. De 5 knapparna i programmet har följande funktion

Välj fil	Välj bitström som ska modifieras.
Ladda ner till FPGA	Modifierar bitströmmen så att ditt program läggs i minnet. Därefter laddas designen ner till FPGA-kortet via USB-kabeln.
Spara som	Sparar programmet och kommentarer i en .hex-fil
Öppna	Hämta program och kommentarer från .hex-fil
Rensa	Nollställer alla instruktioner och kommentarer.

När programmet laddas ner kommer även konfigurationen för hårdvaran laddas över till FPGA-kortet. Du behöver alltså inte själv konfigurera FPGA:n via Vivado.

Labbkortet som är anslutet till Nexys4 består av en trafikkorsning och en lejonbur, se figur 5.10. Med strömställaren S3 väljer man om kortet skall fungera som en trafikkorsning (TRAFIK) eller lejonbur (LEJON). Det finns två kontakter på labbkortet märkta `JA` och `JB`. Dessa är anslutna till motsvarande kontakter på Nexys4.



Figur 5.10: Labbkort

`JB` är signaler från Nexys4 till labbkortet enligt tabellen nedan

JB	**Strömställare S3 i läge trafik**	**Strömställare S3 i läge lejon**
D0 (LSB)	Rött ljus huvudgata	D0 (LSB) till 7-segmentsavkodare
D1	Gult ljus huvudgata	D1 till 7-segmentsavkodare
D2	Grönt ljus huvudgata	D2 7-segmentsavkodare
D3	Rött ljus sidogata	D3 (MSB) 7-segmentsavkodare
D4	Gult ljus sidogata	
D5	Grönt ljus sidogata	
D6		
D7 (MSB)		

`JA` är signaler från labbkort till Nexys4 enligt nedan (oberoende av strömställaren S3):

JA	**Funktion**
D0 (LSB)	0 = Dag, 1 = Natt (S4)
D1	1 = Bil på sidogata (S5)
D2	1 = Lejon framför givare G1
D3	1 = Lejon framför givare G2
D4	0
D5	0
D6	0
D7 (MSB)	0

Uppgift 5.5. Programmering av trafikkorsning

I denna uppgift gäller det att programmera styrningen av trafikljus i en korsning, ett trafikljus på en huvudgata och ett på korsande sidogata. Strömställaren S4 fungerar som omkopplare mellan dag och natt. Knappen S5 används för att simulera att en bil kommer på sidogatan, se figur 5.10.

Under dagtid då S4 = 0 ska trafikljuset fungera enligt följande sekvens som upprepas

Huvudgata	**Sidogata**
grönt	rött
gult	rött
rött	rött
rött	rött + gult
rött	grönt
rött	gult
rött	rött
rött + gult	rött

I läget natt då S4 = 1 ska trafikljuset visa grönt på huvudgatan och rött på sidogatan. Då det kommer en bil på sidogatan (S5 = 1 då knappen Bil trycks ner) ska ljuset skall växla en gång så att det blir grönt på sidogatan och bilen kan passera. Därefter ska det återgå till att vara grönt på huvudgatan.

Det ska gå att se de olika ljussekvenserna då den automatiska klockan på 1 KHz används. Valet mellan 1 KHz och manuell klocka görs med strömställaren `SW15`.

Om kretsen nollställs med knappen `CPU_RESET` och den manuella klockan är vald, måste två klockapulser genereras för att nollställningen ska utföras.

Bitströmmen som ska modifieras är

<sökväg till projektets mapp>\<projektnamn>.runs \impl_1 \mcu.bit

Slut på uppgift 5.5

Uppgift 5.6. Programmering av lejonbur

I denna uppgift skall en kontrollenhet som håller reda på hur många lejon som är ute i en inhägnad programmeras . I princip är denna uppgift identisk med labbuppgift 1.3 med undantagen att det inte finns någon lampa för fara och endast 9 lejon ska räknas.

Slut på uppgift 5.6

För att bli godkänd på laboration 5 krävs att du programmerar er MCU enligt ovanstående uppgifter och kan redovisa i detalj hur ni har löst uppgifterna, både hård- och mjukvara.

APPENDIX A

DATABLAD

Under laborationerna kommer ni att använda logiska kretsar och olika standardkomponenter. Alla dessa finns beskrivna i datablad som tillverkarna ger ut. Databladen till de komponenter som används under laborationerna i Digitalteknik finns i detta appendix. Många komponenter har ganska utförliga datablad där t.ex. tidsförlopp och ström- och spänningsnivåer finns beskrivna. Vi har av utrymmesskäl valt att bara återge de delar som är intressanta för laborationerna.

Innehåll

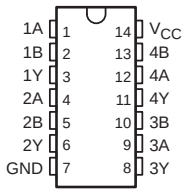
74HC00 4 st 2-ingångars NAND

SN54HC00, SN74HC00
QUADRUPLE 2-INPUT POSITIVE-NAND GATES

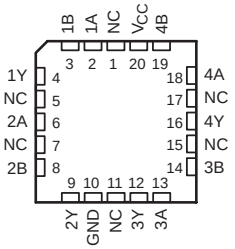
SCLS181B – DECEMBER 1982 – REVISED MAY 1997

- **Package Options Include Plastic Small-Outline (D), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC00 . . . J OR W PACKAGE
SN74HC00 . . . D, N, OR PW PACKAGE
(TOP VIEW)



SN54HC00 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

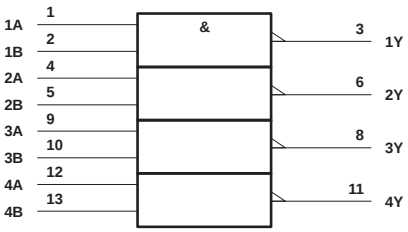
These devices contain four independent 2-input NAND gates. They perform the Boolean function $Y = \overline{A \bullet B}$ or $Y = \overline{A} + \overline{B}$ in positive logic.

The SN54HC00 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC00 is characterized for operation from –40°C to 85°C.

FUNCTION TABLE
(each gate)

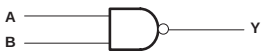
INPUTS		OUTPUT
A	B	Y
H	H	L
L	X	H
X	L	H

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, J, N, PW, and W packages.

logic diagram (positive logic)



TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

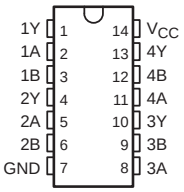
74HC02 4 st 2-ingångars NOR

SN54HC02, SN74HC02
QUADRUPLE 2-INPUT POSITIVE-NOR GATES

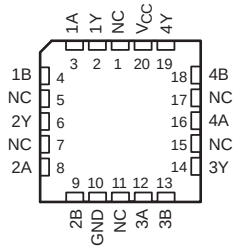
SCLS076B – DECEMBER 1982 – REVISED MAY 1997

- **Package Options Include Plastic Small-Outline (D), Shrink Small-Outline (DB), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC02 . . . J OR W PACKAGE
SN74HC02 . . . D, DB, N, OR PW PACKAGE
(TOP VIEW)



SN54HC02 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

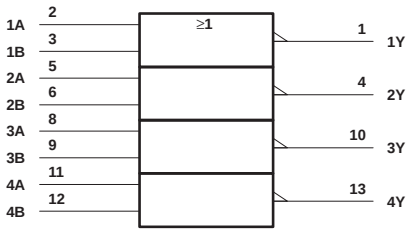
These devices contain four independent 2-input NOR gates. They perform the Boolean function $Y = \overline{A + B}$ or $Y = \overline{A} \bullet \overline{B}$ in positive logic.

The SN54HC02 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC02 is characterized for operation from –40°C to 85°C.

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
H	X	L
X	H	L
L	L	H

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, DB, J, N, PW, and W packages.

logic diagram (positive logic)

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

74HC04 6 st inverterare

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078B – DECEMBER 1982 – REVISED MAY 1997

- **Package Options Include Plastic Small-Outline (D), Shrink Small-Outline (DB), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

description

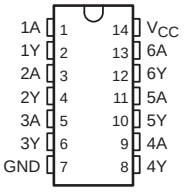
These devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

The SN54HC04 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC04 is characterized for operation from –40°C to 85°C.

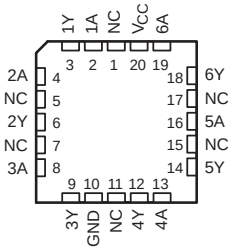
FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, DB, N, OR PW PACKAGE
(TOP VIEW)

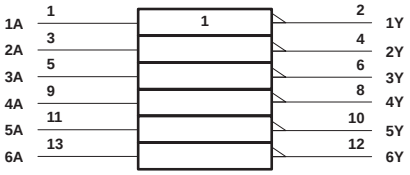


SN54HC04 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, DB, J, N, PW, and W packages.

logic diagram (positive logic)

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

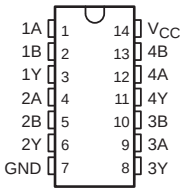
74HC08 4 st 2-ingångars AND

SN54HC08, SN74HC08
QUADRUPLE 2-INPUT POSITIVE-AND GATES

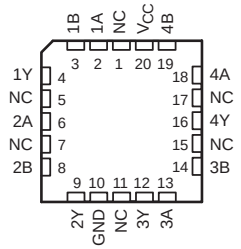
SCLS081B – DECEMBER 1982 – REVISED MAY 1997

- **Package Options Include Plastic Small-Outline (D), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC08 . . . J OR W PACKAGE
SN74HC08 . . . D, N, OR PW PACKAGE
(TOP VIEW)



SN54HC08 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

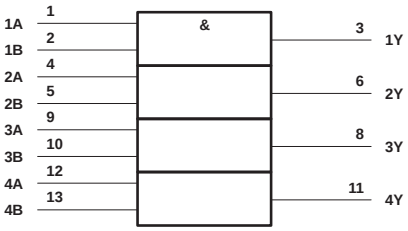
These devices contain four independent 2-input AND gates. They perform the Boolean function $Y = A \bullet B$ or $Y = \overline{\overline{A} + \overline{B}}$ in positive logic.

The SN54HC08 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC08 is characterized for operation from –40°C to 85°C.

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
H	H	H
L	X	L
X	L	L

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, J, N, PW, and W packages.

logic diagram (positive logic)

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

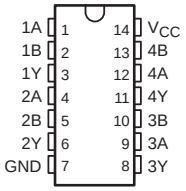
74HC32 4 st 2-ingångars OR

SN54HC32, SN74HC32
QUADRUPLE 2-INPUT POSITIVE-OR GATES

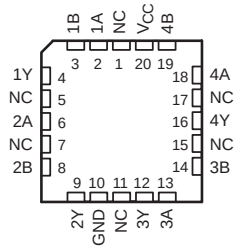
SCLS200B – DECEMBER 1982 – REVISED MAY 1997

- **Package Options Include Plastic Small-Outline (D), Shrink Small-Outline (DB), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC32 . . . J OR W PACKAGE
SN74HC32 . . . D, DB, N, OR PW PACKAGE
(TOP VIEW)



SN54HC32 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

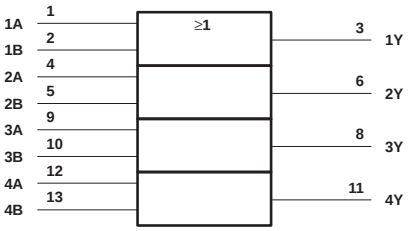
These devices contain four independent 2-input OR gates. They perform the Boolean function $Y = \overline{\overline{A} \bullet \overline{B}}$ or $Y = A + B$ in positive logic.

The SN54HC32 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC32 is characterized for operation from –40°C to 85°C.

FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
H	X	H
X	H	H
L	L	L

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, DB, J, N, PW, and W packages.

logic diagram (positive logic)

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

74HC86 4 st 2-ingångars XOR

SN54HC86, SN74HC86
QUADRUPLE 2-INPUT EXCLUSIVE-OR GATES

SCLS100C – DECEMBER 1982 – REVISED AUGUST 1999

- **Package Options Include Plastic Small-Outline (D) Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) DIPs**

description

These devices contain four independent 2-input exclusive-OR gates. They perform the Boolean function $Y = A \oplus B$ or $Y = \overline{A}B + A\overline{B}$ in positive logic.

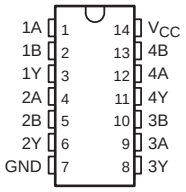
A common application is as a true/complement element. If one of the inputs is low, the other input is reproduced in true form at the output. If one of the inputs is high, the signal on the other input is reproduced inverted at the output.

The SN54HC86 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC86 is characterized for operation from –40°C to 85°C.

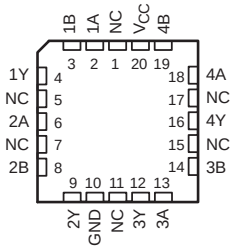
FUNCTION TABLE
(each gate)

INPUTS		OUTPUT
A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

SN54HC86 . . . J OR W PACKAGE
SN74HC86 . . . D, N, OR PW PACKAGE
(TOP VIEW)

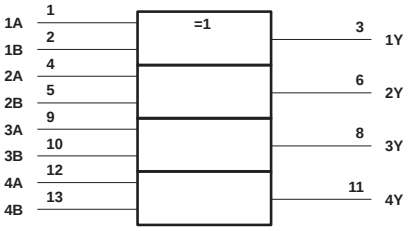


SN54HC86 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, J, N, PW, and W packages.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

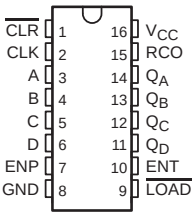
74HC163 4 bitars binärräknare

SN54HC163, SN74HC163
4-BIT SYNCHRONOUS BINARY COUNTERS

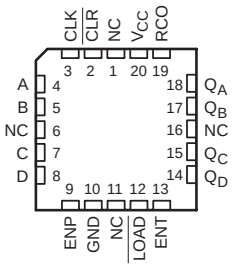
SCLS298A – JANUARY 1996 – REVISED MAY 1997

- **Internal Look-Ahead for Fast Counting**
- **Carry Output for n-Bit Cascading**
- **Synchronous Counting**
- **Synchronously Programmable**
- **Package Options Include Plastic Small-Outline (D) and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC163 . . . J OR W PACKAGE
SN74HC163 . . . D OR N PACKAGE
(TOP VIEW)



SN54HC163 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

These synchronous, presettable counters feature an internal carry look-ahead for application in high-speed counting designs. The ’HC163 are 4-bit binary counters. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincident with each other when instructed by the count-enable (ENP, ENT) inputs and internal gating. This mode of operation eliminates the output counting spikes normally associated with synchronous (ripple-clock) counters. A buffered clock (CLK) input triggers the four flip-flops on the rising (positive-going) edge of the clock waveform.

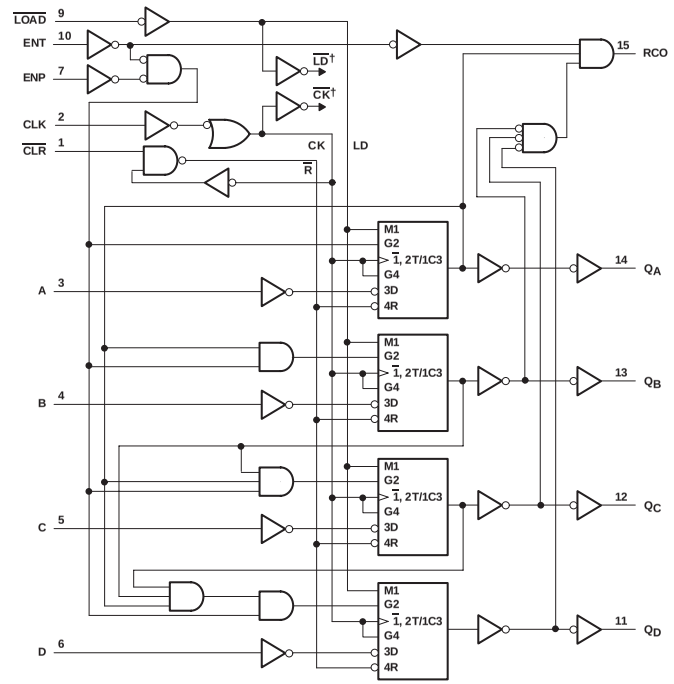
These counters are fully programmable; that is, they can be preset to any number between 0 and 9 or 15. As presetting is synchronous, setting up a low level at the load input disables the counter and causes the outputs to agree with the setup data after the next clock pulse, regardless of the levels of the enable inputs.

The clear function for the ’HC163 is synchronous. A low level at the clear ($\overline{CLR}$) input sets all four of the flip-flop outputs low after the next low-to-high transition of CLK, regardless of the levels of the enable inputs. This synchronous clear allows the count length to be modified easily by decoding the Q outputs for the maximum count desired. The active-low output of the gate used for decoding is connected to $\overline{CLR}$ to synchronously clear the counter to 0000 (LLLL).

The carry look-ahead circuitry provides for cascading counters for n-bit synchronous applications without additional gating. ENP, ENT, and a ripple-carry output (RCO) are instrumental in accomplishing this function. Both ENP and ENT must be high to count, and ENT is fed forward to enable RCO. Enabling RCO produces a high-level pulse while the count is maximum (9 or 15 with Q_A high). This high-level overflow ripple-carry pulse can be used to enable successive cascaded stages. Transitions at ENP or ENT are allowed, regardless of the level of CLK.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

logic diagram (positive logic)

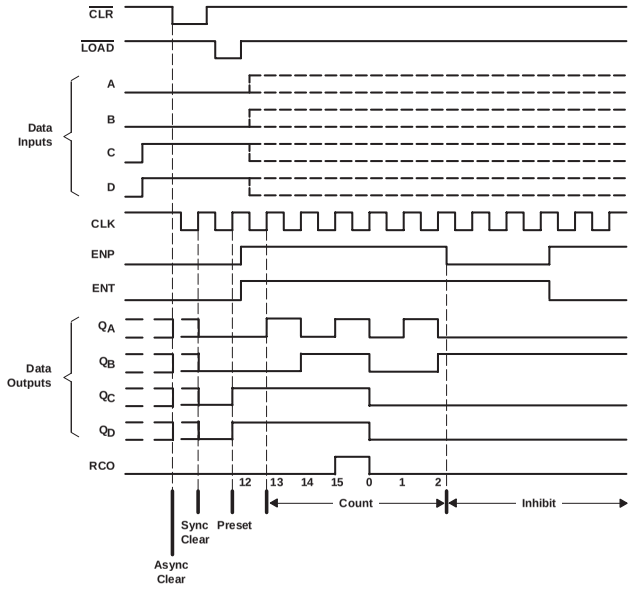


† For simplicity, routing of complementary signals $\overline{LD}$ and $\overline{CK}$ is not shown on this overall logic diagram. The uses of these signals are shown on the logic diagram of the D/T flip-flops.
Pin numbers shown are for the D, J, N, and W packages.

typical clear, preset, count, and inhibit sequence

The following sequence is illustrated below:

1. Clear outputs to zero (synchronous)
2. Preset to binary 12
3. Count to 13, 14, 15, 0, 1, and 2
4. Inhibit



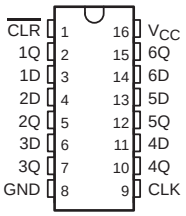
74HC174 6 st D-element

SN54HC174, SN74HC174
HEX D-TYPE FLIP-FLOPS
WITH CLEAR

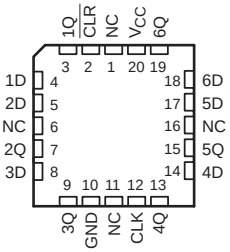
SCLS119B – DECEMBER 1982 – REVISED MAY 1997

- **Contain Six Flip-Flops With Single-Rail Outputs**
- **Applications Include:**
  - **Buffer/Storage Registers**
  - **Shift Registers**
  - **Pattern Generators**
- **Package Options Include Plastic Small-Outline (D) and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC174 . . . J OR W PACKAGE
SN74HC174 . . . D OR N PACKAGE
(TOP VIEW)



SN54HC174 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

These monolithic positive-edge-triggered D-type flip-flops have a direct clear ($\overline{CLR}$) input.

Information at the data (D) inputs meeting the setup time requirements is transferred to the outputs on the positive-going edge of the clock (CLK) pulse. Clock triggering occurs at a particular voltage level and is not directly related to the transition time of the positive-going edge of CLK. When CLK is at either the high or low level, the D input has no effect at the output.

The SN54HC174 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC174 is characterized for operation from –40°C to 85°C.

FUNCTION TABLE
(each flip-flop)

INPUTS			OUTPUT
$\overline{CLR}$	CLK	D	Q
L	X	X	L
H	↑	H	H
H	↑	L	L
H	L	X	Q_0

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

74LS191 4 bitars binär upp/ner-räknare

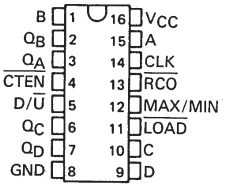
SN54190, SN54191, SN54LS190, SN54LS191, SN74190, SN74191, SN74LS190, SN74LS191
SYNCHRONOUS UP/DOWN COUNTERS WITH DOWN/UP MODE CONTROL

SDLS072 – DECEMBER 1972 – REVISED MARCH 1988

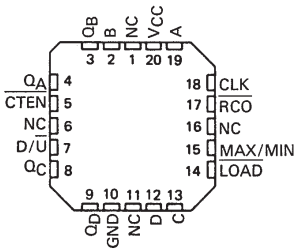
- Counts 8-4-2-1 BCD or Binary
- Single Down/Up Count Control Line
- Count Enable Control Input
- Ripple Clock Output for Cascading
- Asynchronously Presettable with Load Control
- Parallel Outputs
- Cascadable for n-Bit Applications

TYPE	AVERAGE PROPAGATION DELAY	TYPICAL MAXIMUM CLOCK FREQUENCY	TYPICAL POWER DISSIPATION
'190,'191	20ns	25MHz	325mW
'LS190,'LS191	20ns	25MHz	100mW

SN54190, SN54191, SN54LS190, SN54LS191 . . . J PACKAGE
SN74190, SN74191 . . . N PACKAGE
SN74LS190, SN74LS191 . . . D OR N PACKAGE
(TOP VIEW)



SN54LS190, SN54LS191 . . . FK PACKAGE
(TOP VIEW)



NC - No internal connection

description

The '190, 'LS190, '191, and 'LS191 are synchronous, reversible up/down counters having a complexity of 58 equivalent gates. The '191 and 'LS191 are 4-bit binary counters and the '190 and 'LS190 are BCD counters. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincident with each other when so instructed by the steering logic. This mode of operation eliminates the output counting spikes normally associated with asynchronous (ripple clock) counters.

The outputs of the four master-slave flip-flops are triggered on a low-to-high transition of the clock input if the enable input is low. A high at the enable input inhibits counting. Level changes at the enable input should be made only when the clock input is high. The direction of the count is determined by the level of the down/up input. When low, the counter count up and when high, it counts down. A false clock may occur if the down/up input changes while the clock is low. A false ripple carry may occur if both the clock and enable are low and the down/up input is high during a load pulse.

These counters are fully programmable; that is, the outputs may be preset to either level by placing a low on the load input and entering the desired data at the data inputs. The output will change to agree with the data inputs independently of the level of the clock input. This feature allows the counters to be used as modulo-N dividers by simply modifying the count length with the preset inputs.

The clock, down/up, and load inputs are buffered to lower the drive requirement which significantly reduces the number of clock drivers, etc., required for long parallel words.

Two outputs have been made available to perform the cascading function: ripple clock and maximum/minimum count. The latter output produces a high-level output pulse with a duration approximately equal to one complete cycle of the clock when the counter overflows or underflows. The ripple clock output produces a low-level output pulse equal in width to the low-level portion of the clock input when an overflow or underflow condition exists. The counters can be easily cascaded by feeding the ripple clock output to the enable input of the succeeding counter if parallel clocking is used, or to the clock input if parallel enabling is used. The maximum/minimum count output can be used to accomplish look-ahead for high-speed operation.

Series 54' and 54LS' are characterized for operation over the full military temperature range of −55°C to 125°C; Series 74' and 74LS' are characterized for operation from 0°C to 70°C.

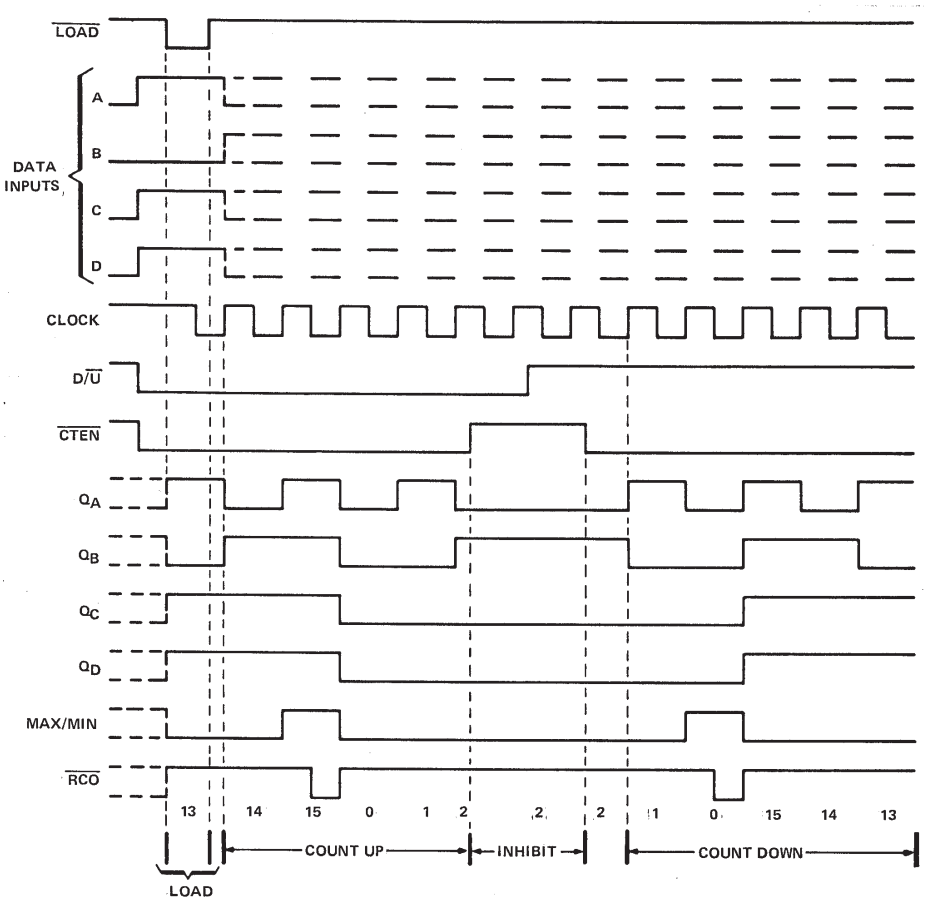
TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

'191, 'LS191 BINARY COUNTERS

pical load, count, and inhibit sequences

Illustrated below is the following sequence:

1. Load (preset) to binary thirteen.
2. Count up to fourteen, fifteen (maximum), zero, one, and two.
3. Inhibit.
4. Count down to one, zero (minimum), fifteen, fourteen, and thirteen.



APPENDIX B

INTRODUKTION TILL VIVADO

Under laborationerna kommer vi att konstruera/beskriva ett antal kretsar med hjälp av VHDL (Very high speed integrated circuit Hardware Description Language). För att skapa verklig hårdvara kommer vi att överföra konstruktionens beskrivning till en FPGA (Field Programmable Gate Array) från tillverkaren Xilinx med hjälp av programmet Vivado, som är Xilinx' utvecklingsmiljö för FPGA:er.

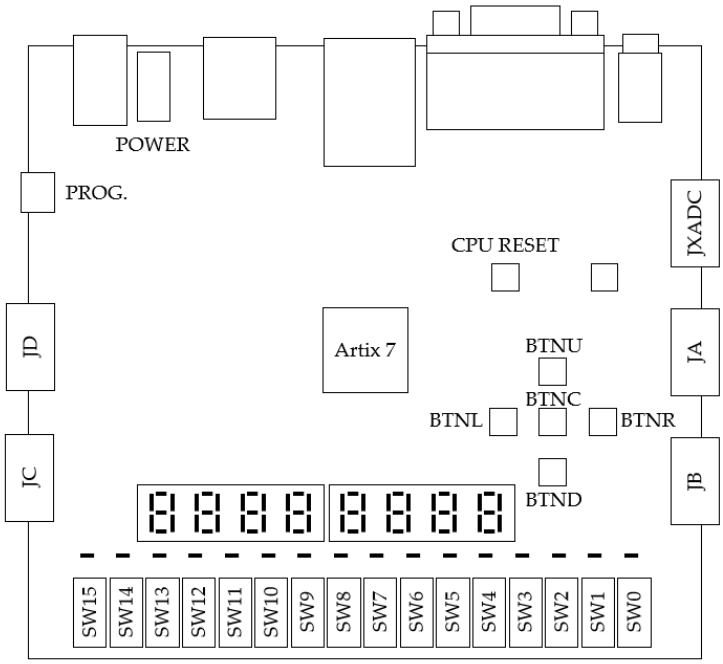
I detta appendix förklaras hur man med Vivado överför en digital konstruktion beskriven i VHDL till en FPGA. Först skapas ett nytt projekt i Vivado. Källfiler som beskriver konstruktionen skapas och läggs till i projektet. Därefter verifieras beteendet hos konstruktionen genom simulering. Efter det följer ett antal steg som syftar till att översätta konstruktionens beskrivning till hårdvara. Slutligen konfigureras FPGA:n så att den beter sig enligt beskrivningen.

För att demonstrera designflödet i Vivado kommer alla stegen för att skapa en enkel konstruktion att gås igenom i detta avsnitt.

Introduktion

En FPGA kan förenklat ses som en komponent innehållande ett stort antal logiska grindar och D-element. Kopplingar mellan de olika grindarna och D-elementen kan helt konfigureras så att önskat beteendet fås.

FPGA:n som används i denna kurs är Artix7 från tillverkaren Xilinx. Den sitter monterad på utvecklingskortet Nexys4, se figur B.1. På utvecklingskortet finns även ett antal kringkomponenter. Vissa av dessa kan användas för att generera insignaler till konstruktionen, som knappar och strömställare, medan lysdioder och 7-segmentsdisplayer kan användas för att visa konstruktionens utsignaler.



Figur B.1: Bild av utvecklingskortet Nexys4 som används i kursen. Ovanför strömställarna *SW0...SW15* sitter lika många lysdioder med namnen *LED0...LED15*.

Konstruktionen av kretsar kommer att göras i programmet Vivado som är utvecklingsmiljön för Xilinx FPGA:er. De interna kopplingarna mellan grindar och D-element i FPGA:n programmeras genom att en konfigurationsfil (*bitstream* i Vivado) laddas ner i FPGA:n från en dator. För att skapa konfigurationsfilen använder vi oss av VHDL som är ett hårdvarubeskrivande språk. Med VHDL beskriver vi hur konstruktionen ska bete sig samt vilka in- och utsignaler den har.

Efter att vi beskrivit en konstruktion i VHDL försöker Vivado översätta beskrivningen till hårdvarukomponenter som grindar och D-element. Resultatet är ett RTL-schema (Register Transfer Level) där man ser hur beskrivningen tolkats. Det går även att simulera konstruktionen och verifiera att den beter sig korrekt.

Efter verifieringen använder vi Vivados syntesverktyg för att översätta RTL-schemat till ett format passande vår målkrets Artix7. I FPGA:n som vi använder finns nämligen inga verkliga grindar utan logiska funktioner implementeras med hjälp av sanningstabeller. Efter syntetiseringen får vi ett schema över konstruktionen bestående av sanningstabeller, muxar, adderare och D-element, samt kopplingarna mellan dessa.

I nästa steg ber vi Vivado att implementera konstruktionen. Det som sker nu är att alla sanningstabeller och andra komponenter får en given position i FPGA:n och att alla ledningar inuti FPGA:n konfigureras så att rätt kopplingar fås.

Efter implementeringen genereras slutligen konfigurationsfilen som sedan laddas ner i FPGA:n via USB.

Konstruktionsexempel

För att visa arbetsgången i Vivado kommer nu alla stegen av en enkel konstruktion att gås igenom. Konstruktionen i exemplet kommer att styra lysdioderna på Nexys4 kortet så att olika ljussekvenser visas. Strömställarna **SW[3:0]** kommer att användas som insignaler till konstruktionen för att bestämma en ljussekvens enligt tabell B.2.

SW_3	SW_2	SW_1	SW_0	**Funktion/ljussekvens**
0	0	0	0	Alla dioder släckta
0	0	0	1	Alla dioder tända
0	0	1	-	Alla dioder blinkar med frekvensen 2Hz
0	1	-	-	Rinnande ljus från vänster till höger och sedan tillbaka
1	-	-	-	Rinnande ljus från sidorna till mitten och sedan tillbaka

Tabell B.2: Beskrivning av konstruktionens funktion. **SW[1]** har högre prioritet än **SW[0]**, **SW[2]** har högre prioritet än **SW[1]** och **SW[3]** har högre prioritet än **SW[2]**.

För att skapa de olika ljussekvenserna kommer vi att använda en i förväg konstruerad modul *led_driver* vars beteende är beskrivet med VHDL i filen *led_driver.vhd*. Modulen har en insignal **C[2:0]** som är 3 bitar bred. Därför måste vi konstruera en krets som avkodar strömställarna och genererar rätt insignal till *led_driver* enligt tabell B.3.

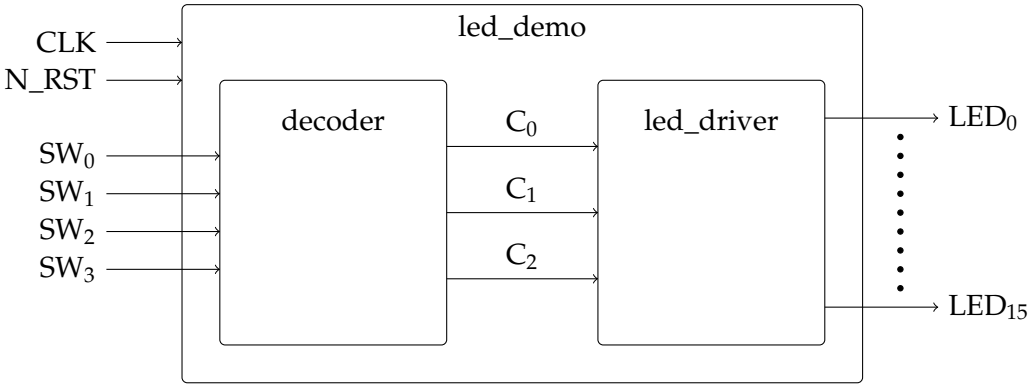
SW_3	SW_2	SW_1	SW_0	C_2	C_1	C_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	0
0	1	-	-	0	1	1
1	-	-	-	1	0	0

Tabell B.3: Tabellen beskriver funktionen hos avkodaren som vi senare ska beskrivas med VHDL.

Utifrån tabellen får vi fram följande uttryck för signalen **C[2:0]**

$$C_0 = (SW_3' \wedge SW_2' \wedge SW_1' \wedge SW_0) \vee (SW_3' \wedge SW_2)$$
$$C_1 = (SW_3' \wedge SW_2' \wedge SW_1) \vee (SW_3' \wedge SW_2)$$
$$C_2 = SW_3$$

Vi kommer att dela upp konstruktionen i tre delar enligt figur B.2. Blocket *led_driver* genererar olika sekvenser till lysdioderna och styrs via blocket *decoder* som läser av strömställarna. Kopplingen mellan blocken, lysdioder och strömställare specificeras i *led_demo*. Vi kommer senare att skapa ett projekt i Vivado där varje block beskrivs i varsin fil. Filnamnen kommer att vara samma som blockens namn med filändelsen *.vhd*, till exempel *led_demo.vhd*.



Figur B.2: Blockschema av konstruktionen. **SW[3:0]** och **LED[15:0]** kommer att anslutas till motsvarande strömställare och lysdioder på Nexys4. **N_RST** kommer att anslutas till knappen *CPU_RESET* och klocksignalen **CLK** till en oscillator på Nexys4.

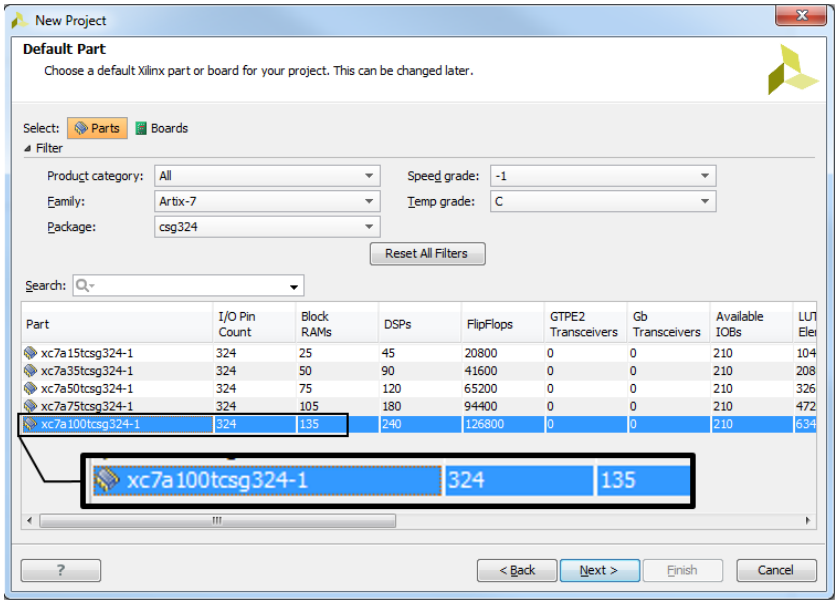
Skapa ett nytt projekt

Vi ska nu börja med att skapa ett nytt projekt i Vivado, lägga till de färdiga filerna *led_driver.vhd* och *led_demo.vhd*, samt skapa filen *decoder.vhd*.

Starta *Vivado 2016.1* genom Windows programmeny. Skapa ett nytt projekt genom att klicka på *Create New Project* alternativt via menyraden *File->New Project...* Guiden för att skapa ett nytt projekt öppnas. Klicka på nästa för att gå vidare och ange projektnamn och var projektet ska sparas. **Projektet måste sparas under "C:\users\<login-id>\Program\" annars kommer inte simuleringar att fungera!**

I nästa steg anges vilken typ av projekt som ska skapas, välj *RTL Project* och se till att *Do not specify sources at this time* är ikryssad.

Därefter måste vi ange vilken FPGA-krets som designen är tänkt att realiseras på. På Nexys4 sitter en Artix7 med beteckningen *xc7a100tcsg324-1*, leta upp den i listan och gå vidare till nästa steg, se figur B.3. Till sist visas en sammanfattning av det nya projektet, klicka på *Finish* för att skapa projektet.



Figur B.3: Vid skapandet av ett nytt projekt måste vi välja vilken FPGA som används. FPGA:n på utvecklingskortet som vi använder har beteckningen xc7a100tcsg324-1.

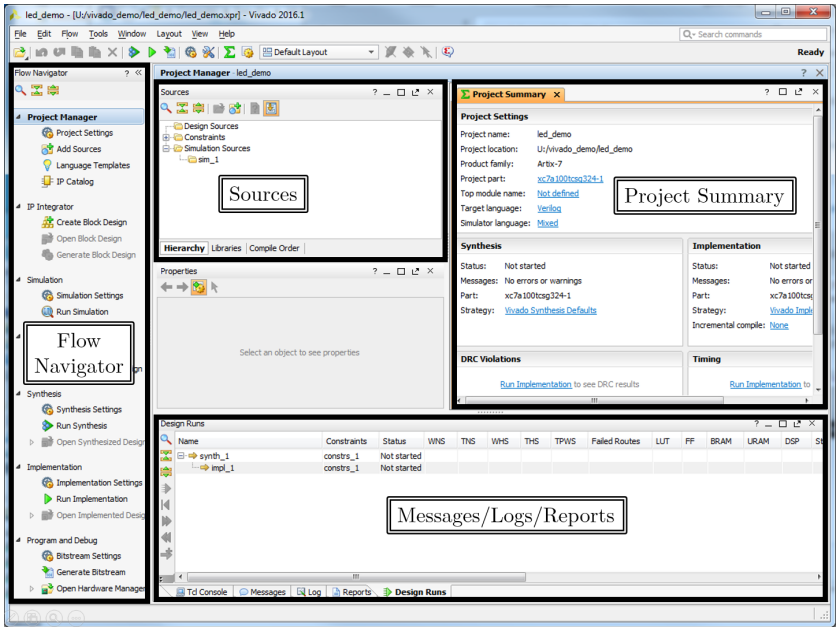
Efter att projektet skapats startas Vivado och ett antal olika fönster visas, se figur B.4. Till vänster har vi *Flow Navigator*. Här kan vi lägga till källfiler, simulera beteendet, syntetisera, implementera, generera konfigurationsfil och överföra konfigurationen till FPGA:n. Det vill säga, härifrån kommer vi åt alla funktioner som behövs för att realisera en design på FPGA:n.

Till höger hittar vi *Project Summary*. Här kan vi se om varningar och fel genererats vid de olika designstegen. Efter att vi utfört syntetisering och implementering blir även information om andel utnyttjad hårdvara och effektförbrukning i FPGA:n tillgänglig här.

I fönstret för *Messages/Logs/Reports* kommer vi åt mer detaljerad information rörande varningar och fel som uppstår. Här finns även en del annan information tillgänglig, så som maximal klockfrekvens för designen med mera.

Efterhand som vi lägger till filer i projektet kommer dessa att listas under *Sources* i olika mappar beroende på vad filen används till.

Om man råkat stänga något fönster eller av någon annan anledning vill återställa vilka fönster som ska visas så görs detta via menyraden *Layout->Reset Layout*.



Figur B.4: Överblicksbild av Vivado efter att ett projekt skapats. Via menyn *Layout->Reset Layout* återgår vi till standardinställningen för vilka fönster som ska visas.

Lägga till och skapa filer

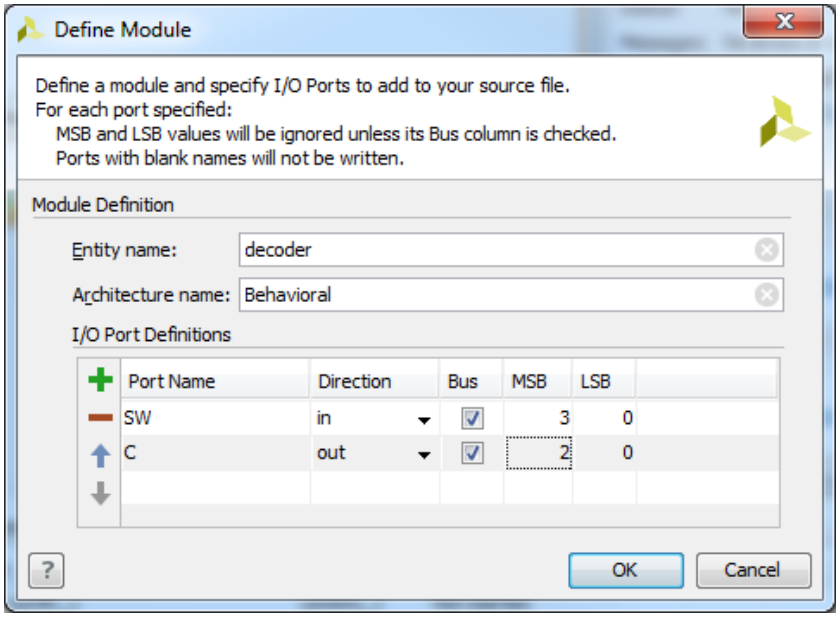
Efter att projekt skapats är det dags att lägga till källfiler som beskriver kretsen vi vill konstruera. Till att börja med så lägger vi till källfiler för *led_driver* och *led_demo*. Därefter skapar vi en ny källfil för *decoder*-modulen och fyller den med kod.

Välj *Add sources* under *Project Manager* i *Flow Navigator*-fönstret. Eftersom vi vill lägga till en källfil för designen väljer vi *Add or create design sources* och går vidare till nästa steg där vi klickar på *Add files*. Gå till mappen *S:\Courses\EIT\EITF65\Vivado_demo* och välj *led_driver.vhd* och *led_demo.vhd*. Se till att rutan **Copy sources into project** är markerad.

Vi ska även skapa en ny fil för *decoder*-modulen. Klicka på *Create File*, se till att *File type* är satt till VHDL och ange *decoder* som *File name*. Klicka på *OK* och därefter *Finish*. Ett nytt fönster öppnas och där fyller vi i vilka in- och utsignaler *decoder*-modulen ska ha. Se till att allt ser ut som i figur B.5 och klicka på *OK*.

Efter att filer lagts till läser Vivado av innehållet och listar alla moduler (entities) i ett hierarkiskt träd under *Sources*. I det här exemplet har vi tre moduler *led_demo*, *decoder* och *led_driver*.

I det här projektet är redan källfiler för de två modulerna *led_demo*, och *led_driver* klara. Vi ska nu beskriva beteendet för den tredje modulen *decoder*. Öppna källfilen för *decoder*-modulen genom att dubbelklicka på modulen under *Sources*. När källfilen öppnats ser vi att Vivado redan har fyllt i de delar som beskriver hur modulens gränssnitt med in- och utsignaler ser ut. Om allt gått rätt till så ska in- och utsignaler enligt figur B.5 finnas med.



Figur B.5: Efter att vi skapat en ny källfil frågar Vivado vilka in- och utsignaler modulen ska ha. Här anger vi att insignalen **SW** är 4 bitar bred och utsignalen **C** är 3 bitar bred.

Beteendet för *decoder*-modulen gavs i tabell B.3. Ändra källkoden så den ser ut som i figur B.6. Glöm inte att spara filen efter modifieringen.

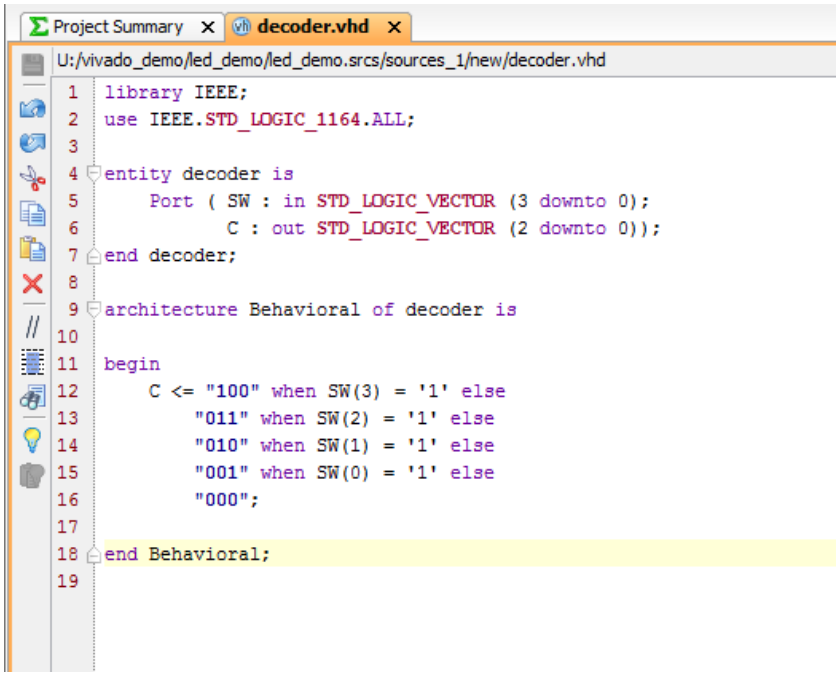
Simulering

Projektet måste vara sparat under "C:\users\<login-id>\Program" annars kommer inte simuleringar att fungera!

För att kontrollera att *decoder* modulen uppför sig korrekt ska vi nu använda Vivados inbyggda simulator för att verifiera att rätt utsignaler genereras vid olika insignaler.

Verifieringen kan göras på lite olika sätt men i den här kursen nöjer vi oss med att använda en testfil som genererar insignaler till modulen vars beteende vi vill studera. Sedan tittar vi på utsignalsvärden och avgör om beteendet är korrekt.

Innan vi kan starta simuleringen måste vi lägga till källkoden för testfilen. Klicka på *Add Sources* och välj alternativet *Add or create simulation sources*. Vivado ser då till att de filer som läggs till endast används under simulering och inte då hårdvara ska genereras. I nästa steg väljer vi *Add Files*, gå till mappen *S:\Courses\EIT\EITF65\Vivado_demo* och välj filen *decoder_tb.vhd*. Se till att alternativet **Copy sources into project** är markerat och klicka på *Finish*.



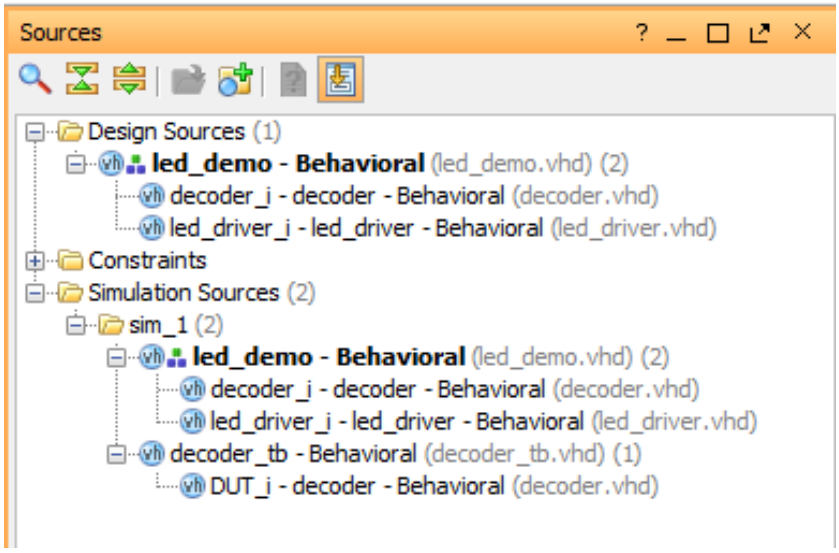
Figur B.6: Källkod för *decoder* modulen som används för att avkoda strömställarna. När Vivado skapat filen innehåller den ett antal rader som börjar med - -. Dessa rader är endast kommentarer och har tagits bort från källkoden som visas.

Filen som importerades finns nu under *Sources->Simulation Sources*, se figur B.7. Här finns även de moduler som beskriver hårdvaran listade (*led_demo*, *decoder* och *led_driver*). I figur B.7 visas *led_demo* i fetstil. Det betyder att simuleringen kommer att utgå från den här modulen när den körs. Men nu vill vi simulera hur *decoder* modulen beter sig. Vi måste därför ställa in att simuleringen ska utgå från *decoder_tb* i stället. Vi ska även ange tiden för hur länge simuleringen ska köras.

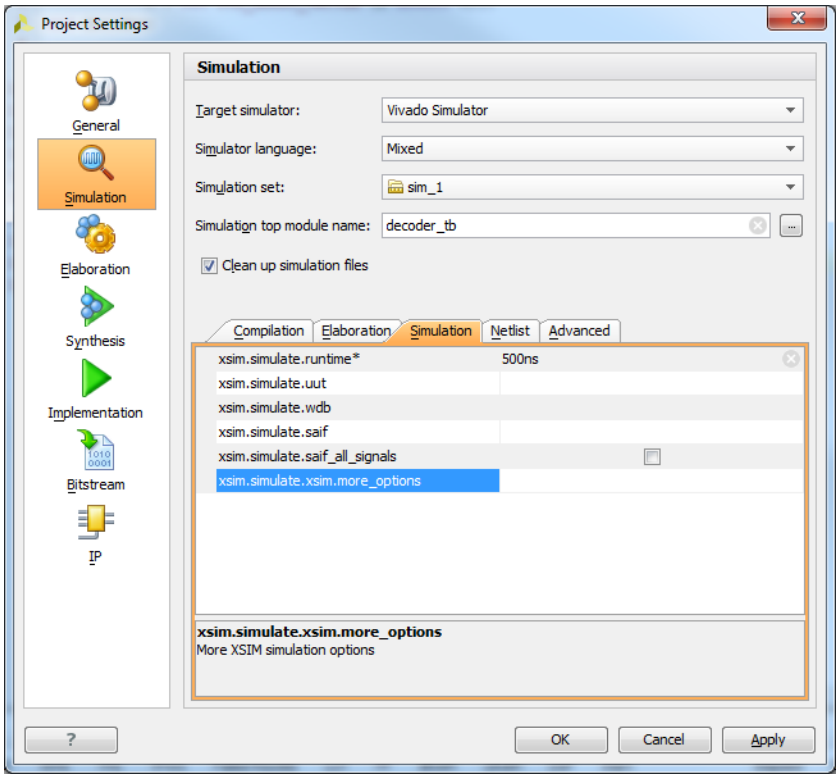
Klicka på *Simulation Settings* i *Flow Navigator*. För att ändra modul som simuleringen ska utgå ifrån klickar vi på ... vid *Simulation top module name* och väljer *decoder_tb*. Gå därefter till fliken *Simulation* och ändra simuleringstiden till 500 ns så att det ser ut som i figur B.8, klicka därefter på *OK*.

Vid större projekt är det vanligt att man har flera olika testfiler som används för att verifiera olika delar av en konstruktion. Man väljer då vilken testfil som simuleringen ska köras ifrån på samma sätt som ovan, under *Simulation Settings->Simulation top module name*. Tiden för hur länge en simulering ska köras beror på hur testfilen är utformad. Det är dock möjligt att förlänga tiden medan simuleringen körs.

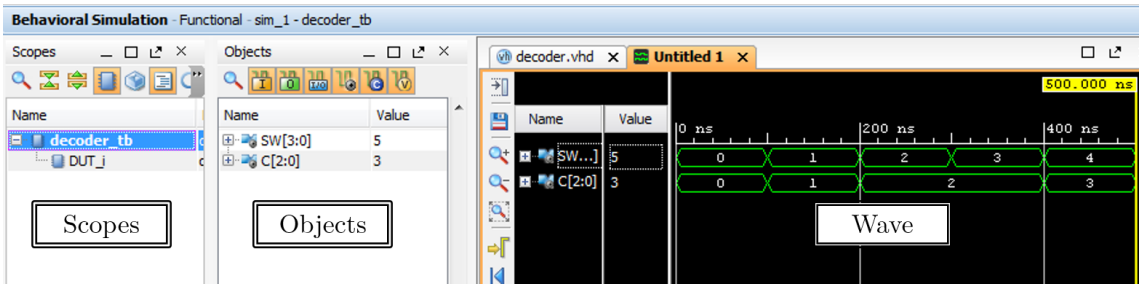
För att starta simuleringen väljer vi *Simulation->Run Simulation->Run Behavioral Simulation* under *Flow Navigator*. Vivado kommer att simulera konstruktionen från tiden 0 ns till 500 ns och generera insignaler till *decoder* modulen enligt beskrivningen i filen *decoder_tb*. Efteråt visas vågformer av signalerna som tillhör testfilen, se figur B.9. Några nya menyknappar kommer även att bli tillgängliga, se figur B.10.



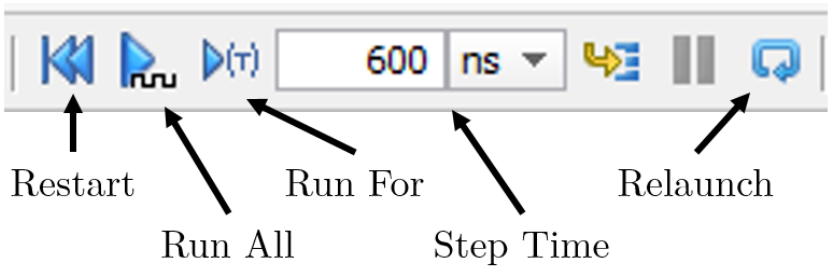
Figur B.7: Filer som är tillgängliga för simulering finns listade under *Simulation Sources* i *Sources*. Simuleringen kommer att utgå från topp-modulen som är markerad i fetstil då den startas.



Figur B.8: Inställningar för Vivados inbyggda simulator. Se till att *Simulation top module name* är satt till *decoder_tb* och *xsim.simulate.runtime* är satt till 500 ns.



Figur B.9: När simuleringen körts visas vågformer för signalerna som tillhör testfilen. Det går även att visa vågformer för signaler som tillhör modulen som testas genom att högerklicka på modulens namn i *Scopes* och välja *Add To Wave Window*. Observera att *decoder*-modulen är instansierad med namnet *DUT_i* i testfilen



Figur B.10: Menyknappar som används för att styra simuleringen i Vivado. *Restart* nollställer simuleringen. *Relaunch* startar om simuleringen och kör den till inställd tid i *Simulation Settings*. *Run All* kör simuleringen tills inga nya insignaler genereras. *Run For* stegar fram simuleringen enligt tiden *Step Time*.

Testfilen som vi kör kommer att generera insignaler till *decoder*-modulen motsvarande de binära talen 0 till 10 med en paus på 100 ns mellan varje tal. Eftersom vi endast körde simuleringen i 500 ns så har vissa insignaler inte genererats än, då simuleringstiden var för kort. Ändra *Step Time* till 600 ns och klicka på *Run for ....*

För att få en bättre överblick av simuleringsresultat högerklickar vi i *Wave* och väljer *Full View* för att se hela tidsförloppet. De enskilda bitarna hos en signal kan visas via plustecknet framför signalens namn i *Wave*.

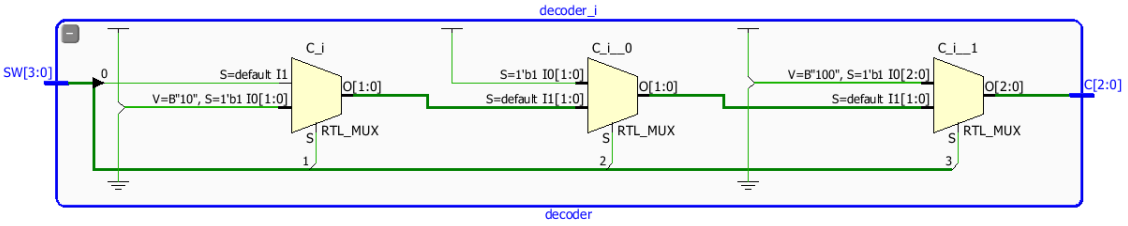
Som standard lägger Vivado till de signaler som tillhör topp-modulen (testfilen) i *Wave*. Genom att markera en annan modul under *Scopes* blir dess signaler synliga i *Objects*. Därifrån kan man högerklicka på en signal och välja *Add to Wave Window*.

Verifiera att *decoder*-modulen beter sig korrekt genom att jämföra simuleringsresultatet med specifikationen i tabell B.3

RTL-schema

Efter att designens beteende beskrivits kan vi se hur Vivado tolkar och översätter detta till hårdvara i ett RTL-schema. Schemat som visas byggs upp med olika logiska block som grindar, muxar, buffrar, adderare och så vidare.

Klicka på *Open Elaborated Design* under *Flow Navigator*. Ett nytt fönster öppnas där vi ser hur de olika modulerna är sammankopplade. Genom att dubbelklicka på *decoder*-blocket går vi ner en nivå och kan se hur avkodarens beteende har översatts till hårdvara, se figur B.11.



Figur B.11: RTL-schema över *decoder* modulen. När RTL-schemat först öppnas visas hela konstruktionen med sina olika block. Genom att dubbelklicka på *decoder*-blocket går vi ner en nivå och kan se RTL-schemat för blocket

Syntetisering

Efter att designen har simulerats och beteendet verifierats är det dags för syntetisering. Det som sker då är att de olika delarna i RTL-schemat översätts till resurser som finns tillgängliga i den aktuella FPGA:n.

I denna kurs använder vi en Artix7 FPGA som innehåller ett stort antal sanningstabeller, D-element, adderare samt ett par andra komponenter. Dock så innehåller den inga grindar som behövs för att forma logiska uttryck. Detta löses under syntetiseringen genom att sanningstabellerna konfigureras så att de realiserar de olika uttrycken i designen.

Innan syntetiseringen kan genomföras måste vi lägga till en fil som talar om för Vivado vilka begränsningar vi har på olika delar i designen. I denna kurs är vi främst intresserade av att bestämma vilka fysiska anslutningspunkter på FPGA:n som designens in- och utsignaler ska anslutas till. I den här konstruktionen vill vi till exempel se till att de fyra strömställarna i det nedre högra hörnet på Nexys4-kortet ansluts till designens insignal **SW[3:0]**.

Till vår konstruktion finns det redan en färdig begränsningsfil som vi ska använda. Denna läggs till i projektet genom att välja *Add Source* under *Flow Navigator*. I fönstret som öppnas väljer vi *Add or create constraints* och sedan *Add Files*. Gå till *S:\Courses\EIT\EITF65\Vivado_demo* och välj filen *Nexys4_led_demo.xdc*. Se till att **Copy sources into project** är markerad och klicka på *Finish*. Begränsningsfilen finns nu med i projektet under *Sources ->Constraints*.

Nu kan designen syntetiseras via *Run Synthesis* under *Flow Navigator*. Om inga fel upptäcks frågar Vivado om implementering ska utföras. Här gör vi bäst i att välja *Cancel* och kontrollerna om några varningar har genererats genom att titta under *Project Summary*. Om fönstret inte är synligt kan det öppnas via *Window->Project Summary*. Under rubriken *Synthesis* kan vi se om varningar genererades. Klickar vi på länken för varningar så visas de i fönstret för *Messages/Logs/Reports*.

Det går bra att strunta i varningar om man vet vad de innebär och vilka konsekvenser de medför.

Efter syntetisering kan vi se att det har genererats två varningar som båda har samma orsak. Klickar vi på länken för varningarna i *Project Summary* visas *[synth 8-3331] desgin top has unconnected port clk* under *messages*. Varningen beror på att klocksignalen till kretsen inte används.

Tyvärr är det inte alltid lätt att lokalisera orsaken till varningar i Vivado. I det här fallet är det dock enkelt och kan ses i RTL-schemat. Öppna filen för modulen *led_demo* och ändra rad 25 från *CLK => N_RST* till *CLK => CLK*. Spara filen och utför en ny syntetisering. Denna gång ska inga varningar eller fel genereras.

Implementering

Under implementering bestämmer Vivado var i FPGA:n som de olika delarna ska placeras samt vilka ledningar som ska användas i FPGA:n och hur de ska kopplas samman.

För att starta implementering väljer vi *Implementation->Run Implementation* under *Flow Navigator*. Efter implementering frågar Vivado om vi vill utföra någon mer funktion. Precis som vid syntetiseringen så ser man eventuella fel och varningar under *Project Summary*.

Programmering av FPGA

Efter implementering har vi en design som kan överföras till FPGA:n. Men för att konfigurera FPGA:n med implementeringsresultatet måste vi generera en konfigurationsfil som kan överföras till FPGA:n via USB.

För att generera konfigurationsfilen väljer vi *Program and Debud->Generate Bitstream* under *Flow Navigator*. När processen är klar återstår bara att skicka över filen till FPGA:n.

Se till att Nexys4-kortet är anslutet till datorn via USB-kabel. Se till att strömställaren *Power* är i läge *ON*. Välj *Program and Debug->Open Hardware Manager->Open Target* under *Flow Navigator*. I rutan som visas väljer vi *Auto Connect*, Vivado kommer då att leta efter anslutna enheter. Därefter skickar vi över konfigurationsfilen genom *Program and Debug->Hardware manager-> Program Device* under *Flow Navigator* och väljer *xc7a100_0* i rutan som visas.

När FPGA:n blivit programmerad kan vi verifiera att hårdvaran beter sig enligt tabell B.2.

Sammanfattning

Slutligen följer här en liten sammanfattning av designflödet i Vivado som kan användas som referens vid senare projekt.

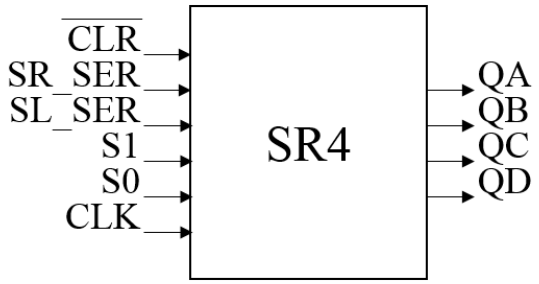
1. Skapa ett nytt projekt i Vivado. Se till att välja rätt FPGA.
2. Lägg till/skapa källfiler som beskriver hårdvaran.
3. Lägg till/skapa källfiler för testmoduler.
4. Simulera och ändra källfiler iterativt tills konstruktionen beter sig korrekt.
5. Lägg till/skapa begränsningsfilen.
6. Syntetisera designen. Åtgärda eventuella fel och varningar. RTL-schemat kan vara till hjälp.
7. Implementera designen. Åtgärda eventuella fel och varningar.
8. Generera konfigurationsfilen.
9. Programmera FPGA:n.
10. Verifiera att hårdvaran beter sig korrekt.
11. Om hårdvaran inte beter sig korrekt kanske beskrivningen av hårdvaran är felaktig. Kanske upptäcktes inte fel vid simuleringen. Eller så struntade någon i varningar som genererades vid syntetiseringen och implementeringen.

APPENDIX C

DATABLAD VHDL

Innehåll

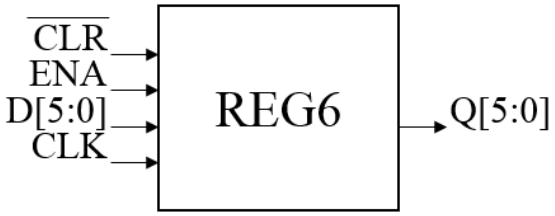
SR4 4-bitars skiftregister



Insignaler						Utsignaler			
$\overline{CLR}$	MODE		CLK	SERIAL		QA	QB	QC	QD
	S1	S0		SL_SER	SR_SER				
0	X	X	↑	X	X	0	0	0	0
1	0	1	↑	X	1	1	QA_0	QB_0	QC_0
1	0	1	↑	X	0	0	QA_0	QB_0	QC_0
1	1	0	↑	1	X	QB_0	QC_0	QD_0	1
1	1	0	↑	0	X	QB_0	QC_0	QD_0	0
1	0	0	↑	X	X	QA_0	QB_0	QC_0	QD_0

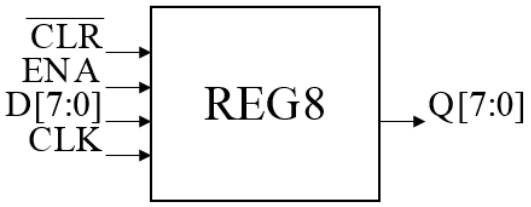
Qx_0 anger tidigare värde/tillstånd på signalen Qx.

REG6 6-bitars register



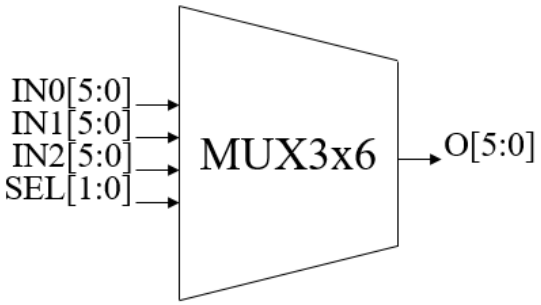
$\overline{CLR}$	ENA	CLK	Q[5:0]
0	X	↑	000000
1	0	X	$Q[5:0]_0$
1	1	0	$Q[5:0]_0$
1	1	↑	D[5:0]

REG8 8-bitars register



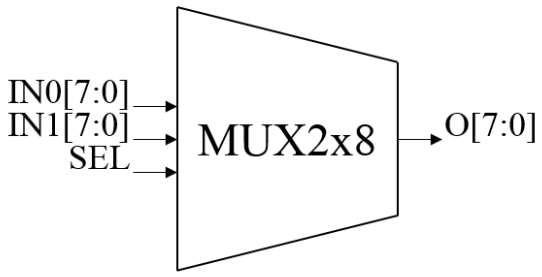
$\overline{CLR}$	ENA	CLK	Q[7:0]
0	X	↑	00000000
1	0	X	$Q[7:0]_0$
1	1	0	$Q[7:0]_0$
1	1	↑	D[7:0]

MUX3x6 6-bitars 3 till 1 multiplexer



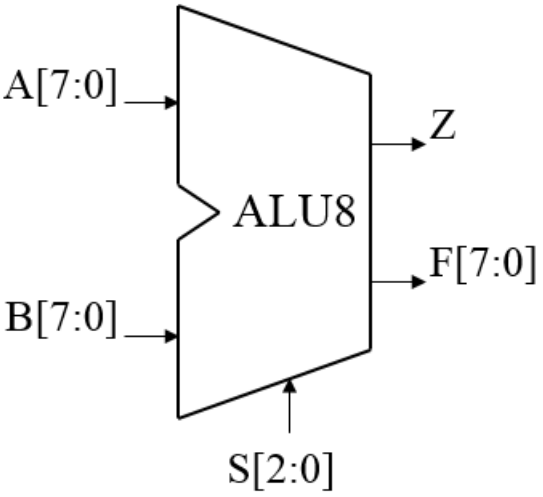
SEL[1:0]	O[5:0]
00	IN0[5:0]
01	IN1[5:0]
10	IN2[5:0]
11	X

MUX2x8 8-bitars 2 till 1 multiplexer



SEL	O[7:0]
L	IN0[7:0]
H	IN1[7:0]

ALU8 8-bitars ALU



S2	S1	S0	Funktion
0	0	0	F=A
0	0	1	F=B
0	1	0	F=A+B
0	1	1	F=B−A
1	0	0	F=A∧B
1	0	1	F=A∨B
1	1	0	F=A⊕B
1	1	1	F=0