LABORATION 5

MCU

Laborationens syfte

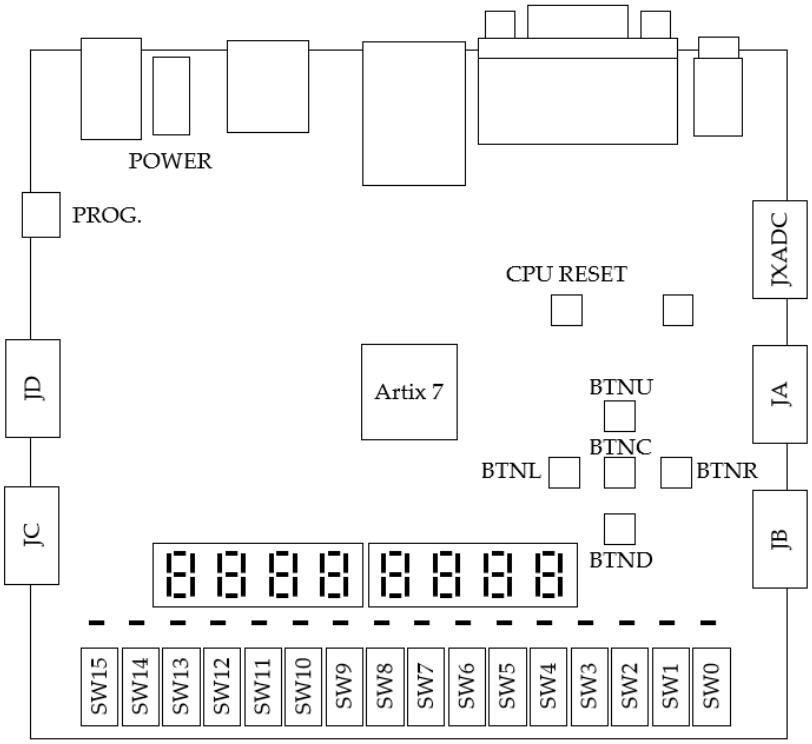
Läs igenom hela laboration 5 innan du börjar beskriva instruktionsavkodaren i VHDL!

I denna laboration ska en enkel MCU (Micro-Controller_Unit) konstrueras. En MCU, eller enkretsdator, kan liknas med en liten 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 enkelt kan anslutas.

Med en MCU kan samma krets användas för att lösa olika problem genom att olika program laddas in i dess minne.

I denna laboration ska en MCU konstrueras med hjälp av VHDL och utvecklingsverktyget Vivado från Xilinx. Hårdvaran realiseras på Nexys4 FPGA-kort. I laboration 6 ska två mikroprogram skrivas som löser uppgifterna lejon och trafik med hjälp av MCU'n.

Nexys4 FPGA utvecklingskort

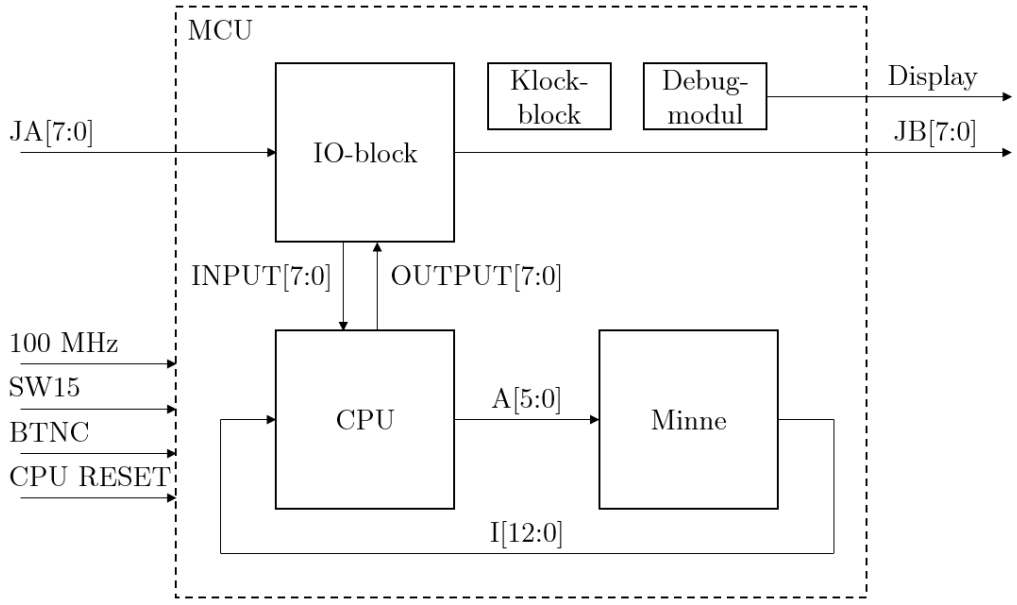


Figur 5.1: Strömställarna `SW0` till `SW15` genererar en logiskt etta då de är i sitt övre läge. Knapparna som är arrangerade i ett kors, `BTNx`, genererar en logiskt etta då de trycks ner. Knappen `CPU RESET` fungerar på motsatt sätt, nolla genereras då den trycks ner. Anslutningarna `JA–JD` är generella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

Introduktion

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 MCUn kommer att kunna utföra ett antal olika operationer, exempelvis addera två tal och spara resultatet. MCUn styrs genom att ett program laddas ner i dess minne. Programmet kommer att bestå av en lista med instruktioner som talar om för MCUn vilka operationer den ska utföra och i vilken ordning.

Efter att hårdvaran har implementerats ska två olika program skrivas och laddas ner i MCUn för att lösa två praktiska problem (Laboration 6). Först ska trafikljusen vid en vägkorsning styras och därefter ska antalet lejon i en bur räknas (likt lejonburen i tidigare laboration).



Figur 5.2: Blockschema av MCUn som ska konstrueras. Observera att klock- och resetsignal inte är utritade.

Hårdvarudelen av laborationen består av att konstruera den del avCPUn (Central-Processing-Unit) som betecknas `instruktionsavkodare` då övriga block i figur 5.3 tillhandahållas som färdiga VHDL moduler. För att kunna fullfölja denna uppgift måste du ha en god uppfattning hur processorn fungerar. Läs igenom nedanstående text noga så att du har en klar uppfattning hur det är tänkt att MCU'n ska fungera.

Uppgift 5.1. Hur MCU'n är uppbyggd

MCUns minne är av typen ROM (Read-Only-Memory) därför kan kretsen inte modifiera dess innehåll utan endast läsa av det. Programmet som styr MCUn 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 i laboration 6.

När ett program körs kommer debug-blocket att skriva ut minnesadressen till instruktion som exekveras för stunden på 7-segmentsdisplayen. Detta kan vara användbart vid felsökning av programmet .

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 5.1. Varje instruktion i programmen kommer att ta en klockcykel för MCUn att utföra.

Externa komponenter som lysdioder och brytare kommer även att anslutas till MCUn under laborationen. Komponenter som genererar insignaler till kretsen ansluts till kontakten `JA` och komponenter som ska visa utsignaler från kretsen asnluts till `JB`. Både `JA` och `JB` består av åtta bitar och synkroniseras med resten av kretsen i IO-blocket (Input-Output). De synkroniserade versionenerna av `JA` och `JB` är signalerna `INPUT` respektive `OUTPUT` i figur 5.2.

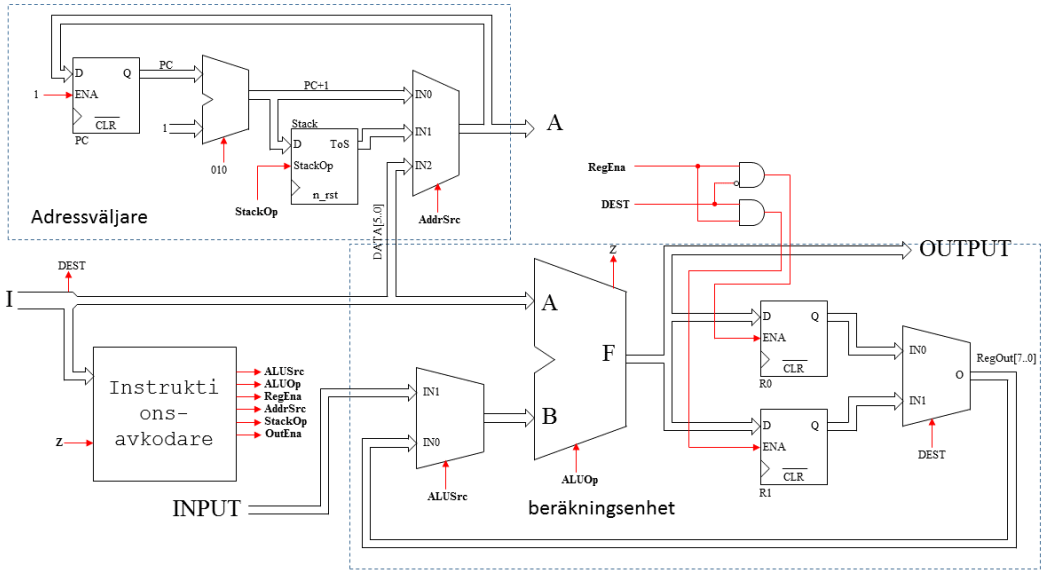
Då ett program exekveras läser CPUn av en instruktion `I` från en adress `A` i minnet, se figur 5.2. CPUn avkodar instruktionen, utför nödvändiga operationer och genererar adressen till nästa instruktion som ska läsas av ifrån minnet.

För att ett program ska exekveras korrekt måste CPUn på något sätt komma ihåg var i programmet den befinner sig. CPUn ska därför ha en programräknare `PC` (Program-Counter), som är ett register[1] vars värde alltid är adressen till den instruktion som körs för stunden, se figur 5.3. Stacken i figuren kommer att förklaras senare men kan ses som ett minne där programadresser kan sparas undan för att senare och läsas ut.

Beräkningar som utförs i CPUn kommer att göras med en ALU. Värdet av utsignalen `F` från ALUn kommer att kunna sparas i ett av de två register `R0` och `R1` i registerblocket. Den ena operanden till ALUn kommer alltid att komma från instruktionen (minnet), den andra operanden kommer att vara en av `R0`, `R1` eller signalen `INPUT`. Konstruktionen blir därmed begränsad i avseende på att det inte går att utföra några operationer mellan de båda registerna `R0` och `R1`.

Instruktionsavkodaren i CPUn har till uppgift att utifrån instruktionen `I` generera styrsignaler till **alla** komponenter i CPUn så att rätt operation erhålls.

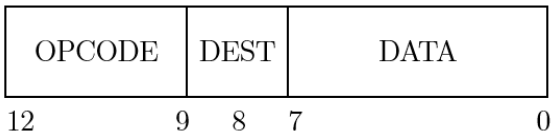
[1] Ett register består av ett antal D-element som sitter parallellt. Register används då man behöver ett minne som kan spara mer än en bit. T.ex. består ett 8-bitars register av åtta D-element och har 2^8 olika tillstånd.



Figur 5.3: Abstrakt blockschema av CPUn.

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. CPUn som konstrueras under laborationen kommer endast stödja ett fåtal instruktioner men kan enkelt byggas ut för att stödja fler.



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

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.

Nedan följer en beskrivning av de instruktioner som CPUn ska stödja. Istället för 4-bitars `OPCODE` anges instruktionerna med förkortningar för att det ska vara lättare att komma

ihåg dem. I listan nedan gäller att

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

`NOP` No OPeration, gör inget förutom att räkan upp adressräknaren `PC+1`.

`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, Z-flaggan = 1). 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 med `RX` och `DATA`.
`RX[i]`$^+$`=RX[i]`$\wedge$`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`.

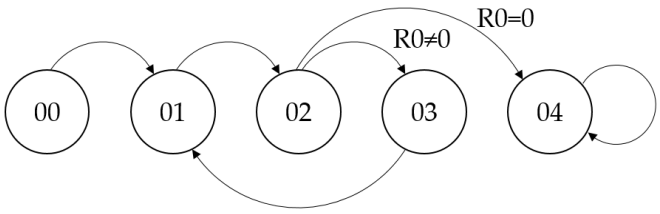
Med instruktionerna ovan kan ett enkelt program skrivas som

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

På progammadress 00 laddas `R0` med värdet 2, därefter minskas `R0`s värde 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.

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

Programmflödet 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 programmflödesdiagram.

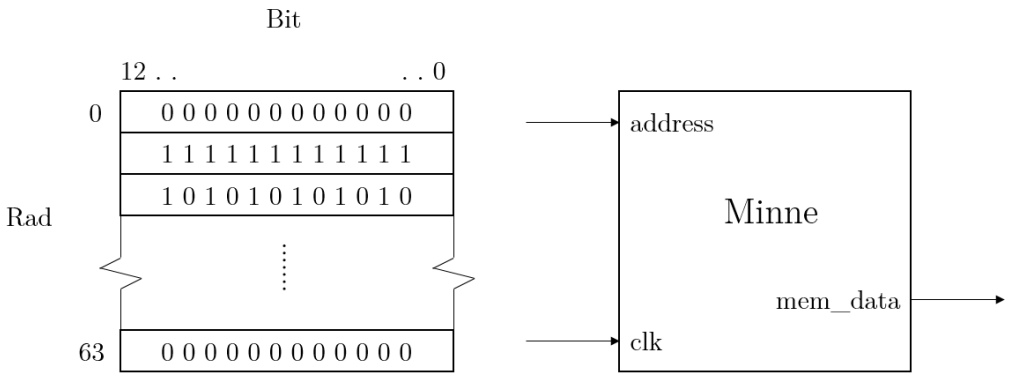
Då minnet som används under laborationen kommer att ha begränsad kapacitet kan det vara bra att återanvända instruktionsstycken. I programmet nedan används instruktionerna på adress 10 till 14 som en subrutin för att generera en fördröjning.

```
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 attt 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 programmrad lagras på varsin minnesrad.

Om vi tittar lite på CPUn ser vi att den är uppdelad i tre huvuddelar, adressväljare, beräkningsenhet och instruktionsavkodare enligt figur 5.3.

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

Instruktionsavkodaren avkodar instruktionen `I` från minnet. Beroende på instruktion och signalen `Z` genereras styrsignaler till alla delar av CPUn 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.

Adressväljaren

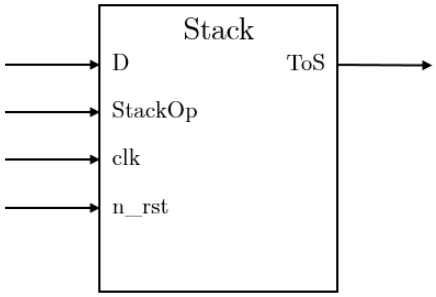
Beroende på instruktionen som exekveras kommer nästa adress att genereras på olika sätt. Det är adressväljarens uppgift att utifrån styrsignaler från instruktionsavkodaren generera rätt adress till minnet.

Vid subrutinsanrop med instruktionen `CALL` så måste återhoppsadressen sparas undan. Detta eftersom CPUn 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).

Stacken består av 4 stycken minnesceller som kan lagra upp till fyra adresser. 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 5.7 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 synkron aktiv låg resetsignal som nollställer stackens celler. Signal `StackOp` ska styra stacken enligt tabellen nedan



Figur 5.7: Symbol av stacken som ska implementeras. Anslutningarna D och ToS är 6 bitar breda och StackOp 2 bitar bred.

`StackOp [1:0]`	Funktion
PUSH , StackOp[01]	Värdet av D sparas överst på stacken. tidigare sparade värdet trycks ned ett steg
POP , StackOp[10]	Stackens innehåll flyttas upp ett steg. och den understa cellen får värdet 0.
HOLD , StackOp[00]	Stackens innehåll oförändrat

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

`StackOp` = PUSH
`D` = 20

	Före:	Efter:
`ToS`	2	20
	0	2
	0	0
	0	0

`StackOp` = POP

	Före:	Efter:
`ToS`	1	2
	2	3
	3	4
	4	0

`StackOp` = HOLD

	Före:	Efter:
`ToS`	1	1
	2	2
	3	3
	0	0

Observera att `StackOP [1:0]` är två av bitarna som instruktionsavkodaren skall hantera!

Stacken ska nu användas för att adressväljaren ska generera rätt adress `A` till minnet vid de olika instruktionerna.

Vid instruktioner som inte påverkar programflödet som `ADD` och `OUT` ges nästa adress av den som följer i ordningen `A = PC+1`.

Vid progamhoppsinstruktionen `B` ska däremot nästa adress vara den som ges av de 6 minst signifikanta bitarna av instruktionen, `A=DATA[5:0]`. Detsamma gäller vid instruktionen `BZ` om resultatet från ALU är noll. Annars ges nästa adress av `A=PC+1`

Även vid subrutinsanrop med instruktionen `CALL` gäller att `A=DATA[5:0]`. Dock så måste återhoppsadressen `PC+1` sparas på stacken. Vid instruktionen `RET` ges nästa adress av det översta elementet på stacken `ToS`.

Insignaler till adressväljarblocket är `clk`, `n_rst`, `StackOp[1:0]`, `AddrSrc[1:0]` och `DATA[5:0]`. Utsignaler är `A` och `pc_debug` som båda är 6 bitar breda.

För att generera `A` kommer det att behövas en komponent som ser till att ett av värdena i tabellen nedan tilldelas `A` beroende på styrsignalen `AddrSrc`.

Nästa adress `A`	AddrSrc[1:0]	**Beskrivning**
`PC+1`	00	Nästa adress i ordningen.
`ToS`	01	Det översta värdet på stacken (Top-of-Stack).
`DATA[5:0]`	10	De 6 minst signifikanta bitarna i instruktionen.

Beräkningsenhet

Beräkningsenheten ska förutom ALUn innehålla registerna `R0` och `R1` (se figur 5.3) vilket är det enda minnet i MCUn där temporär data kan sparas.
ALUn (figur 5.3) är den enhet som utför beräkningar och logiska operationer mellan ingångarna **A** (8-bit) och **B** (8-bit) vars resultat hamnar på utgång **F** (8-bit). Vilken operation som skall utföras bestäms av `AluOp` och är på tre bitar då det finns åtta olika operationer som ALUn kan utföra. Operationerna är enligt nedanstående tabell:

operation	AluOp[2:1:0]	Beskrivning
A	000	F=A
B	001	F=B
A+B	010	F=A+B
A-B	011	F=B-A
A $\wedge$ B	100	F=A and B
A $\vee$ B	101	F=A or B
A $\oplus$ B	110	F= A xor B
0	111	F=0

Registerblocket styrs med de två signalerna `RegEna` och `DEST`. Signalen `DEST`, bit 8 från instruktionen, bestämmer vilket av de två registerna är aktivt. Om `DEST=0` ska värdet av `R0` finnas på registerblockets utsignal `RegOut`, annars ska värdet av `R1` finnas på utsignalen.

Om `RegEna=1` ska det aktiva registrets innehåll uppdateras med resultatet `F` från ALUn , annars ska det aktiva registret vara oförändrat.

Beräkningsenheten har två utsignaler `OUTPUT` och `Z`. Båda dessa genereras från ALUn. Operand `A` är alltid de åtta minst signifikanta bitarna ur instruktionen `DATA[7:0]`. Operand `B` ska däremot antingen komma från det aktiva registerblocket `RegOut` eller från signalen `INPUT` från IO-blocket. Valet av `B` görs med signalen `ALUSrc` där ett högt värde medför att `B=INPUT` och ett lågt värde att `B=RegOut`.

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

Instruktionsavkodare

Instruktionsavkodaren, det sista blocket av CPUn 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 beskrivits 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.

Tabellen nedan är en sammanställning över de olika operationer och signaler med respektive namn och kod som skall användas i hemuppgiften 5.1.1. Om ni använder namn eller kod spelar ingen roll då bägge är definierade. Observera att de olika opcoderna börjar med namnet `OPDODE_`, exempelvis CALL blir `OPCODE_CALL` i VHDL. Naturligtvis går det lika bra med binärkoden för respektive opcode.

Namn	kod
STACK_OP_POP	10
STACK_OP_PUSH	01
STACK_OP_HOLD	00
ADDR_PC_PLUS_ONE	00
ADDR_ToS	01
ADDR_DATA	10
ALUOP_A	000
ALUOP_B	001
ALUOP_A_PLUS_B	010
ALUOP_B_MINUS_A	011
ALUOP_A_AND_B	100
ALUOP_A_OR_B	101
ALUOP_A_MOD_B	110
ALUOP_NULL	111

Hemuppgift 5.1.1

Fyll i tabellen nedan så att instruktionerna implementeras enligt specifikationen tidigare i manualen.

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` visa sig vara mycket användbar senare i laborationen.

OPCODE	I[12:9]	Z	StackOp	AddrSrc	ALUOp	ALUSrc	OutEna	RegEna
NOP	0000	-						
CALL	0001	-						
RET	0010	-						
BZ	0011	0						
BZ	0011	1						
B	0100	-						
ADD	0101	-						
SUB	0110	-						
LD	0111	-						
IN	1000	-						
OUT	1001	-						
AND	1010	-						

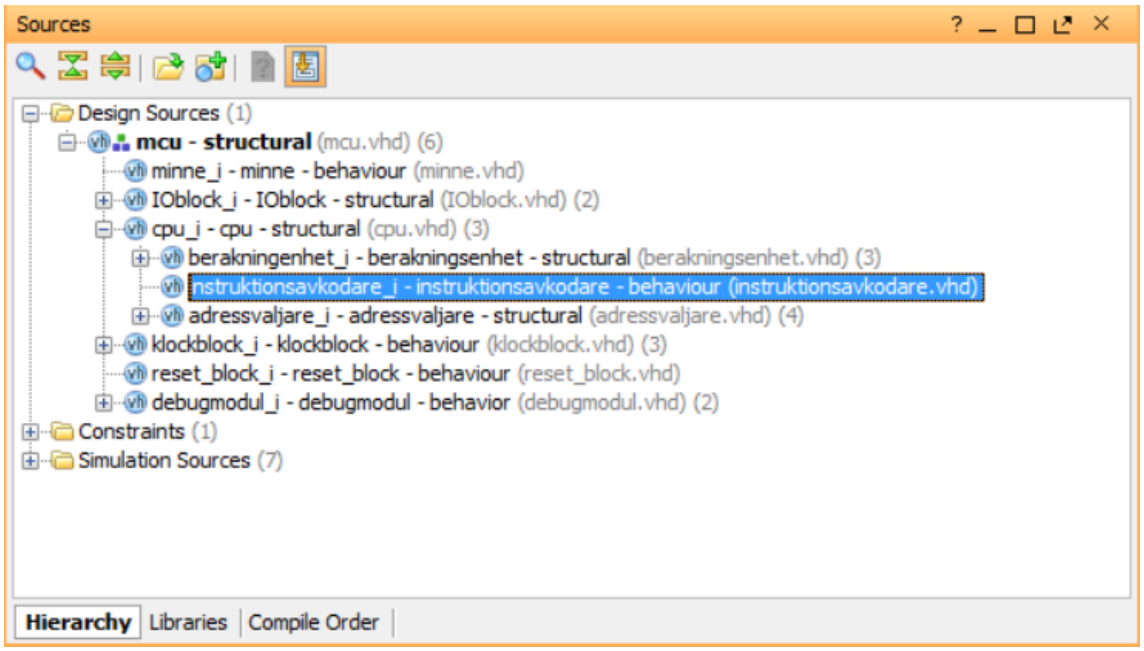
`StackOp` = { PUSH, POP, HOLD }
`AddrSel` = { PC+1, ToS, DATA[5:0] }
`AluOp` = { A, B, A+B, B-A, A $\wedge$ B, A $\vee$ B, A $\oplus$ B , 0}

Slut på uppgift 5.1

Nu har ni förhoppningsvis fått en klar bild av hur MCUn är tänkt att fungera och hur instruktionsavkodaren som spindel i nätet ska styra de övriga delarna i MCUn.
Vi skall nu implementera instruktionsavkodaren i VHDL enligt tabellen som har blivit ifylld enligt hemuppgiften ovan.

Uppgift 5.2. Implementering av instruktionsavkodaren i VHLD

Hela MCUn består av ett antal filer som tillsammans med Xilinx Vivado finns på `S:\courses\eit\EITA15\Lab5` Kopiera hela katalogen (lab5) till `C:\Users\stilID\program\` och ni kommer då att få en nästan komplett miljö för att generera en MCU i VHDL.
Starta Vivado genom att klicka på `labMCU.xpr`.
Det är endast en fil som behöver editeras och det är `instruktionsavkodare.vhd`
OBS! Övriga filer får ej ändras!
`Instruktionsavkodare.vhd` hittar ni i fönstret Sources enligt figur 5.8



Figur 5.8: Instruktionsavkodare.vhd

Fyll i koden som saknas enligt hemuppgiften men nu översatt till fungerande VHDL-kod. När ni är klara med detta ska ni testa `Instruktionsavkodare` genom att köra insttruktionsavkodare_tb. När ni får resultatet OK ta kontakt med handledare och förklara vad ni gjort.
(Se Lab 1 om ni är osäkra på simulering)

Slut på uppgift 5.2

Uppgift 5.3. Implementering av en extra operand, DOUT

Att skicka et värde till utenheten kan upplevas som omständigt. Det hade underlättat om det är möjligt att skriva till OUTPUT utan att behöva involvera något register, det vill säga en operand DOUT som hanterar DATA (från instruktioner) direkt till UT.
Implementera denna instruktion (valfri binärkod) på lämpligt sätt. För att detta ska fungera behöver ni även lägga till lite i filen cpu_pkg.vhd(hittas i fliken library i projektmanager.)

Slut på uppgift 5.3

Uppgift 5.4. Syntetisering och implementering av MCU

Syntetisera, implementera och generera bitströmmen för projektet i Vivado. Bit-filen ligger i katalogen `C:/Users/Stil-ID/program/lab5/lab5.runs/impl_1/mcu.bit`. Kopiera denna mcu.bit till katalogen test_DOUT och titta i laboration 6 hur man gör för att ladda minnet med program.
Skriv ett mycket enkelt program som testar DOUT, typ tänd en lysdiod för att verifiera att MCUn fungerar.

Slut på uppgift 5.4