

Datorkonstruktion TSEA83

Fö 11

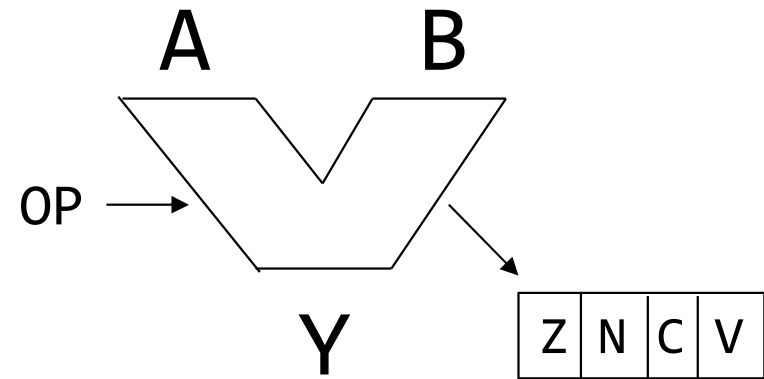
ALU

Datorteknik Fö11 : Agenda

- ALU
- Adressavkodning
- Avbrott
- Subrutiner
- IO-manager
- Hierarki
- Arbetsgång
- Designskiss : Tips : Boot-loader, timing.
- Statusrapport

ALU

En ALU har två uppgifter:



Utföra en operation

- Aritmetik
add, sub, mul, div
- Logiska operationer
and, or, xor, not
- Annan bitmanipulering

Sätta statusflaggor

- Z : Zero flag
- N : Negative flag
- C : Carry flag
- V : Overflow flag
- Andra flaggor

ALU : add, sub

Flaggorna Z och N

Bin	2-kompl	Decimal	Z	N
0000	0	0	1	0
0001	1	1	0	0
0010	2	2	0	0
0011	3	3	0	0
0100	4	4	0	0
0101	5	5	0	0
0110	6	6	0	0
0111	7	7	0	0
1000	-8	8	0	1
1001	-7	9	0	1
1010	-6	10	0	1
1011	-5	11	0	1
1100	-4	12	0	1
1101	-3	13	0	1
1110	-2	14	0	1
1111	-1	15	0	1

4-bitars tal Y:

$$Y = \{y_3, y_2, y_1, y_0\}$$

Flaggorna Z och N:

$$Z = \bar{y}_3 \cdot \bar{y}_2 \cdot \bar{y}_1 \cdot \bar{y}_0$$

$$N = y_3$$

ALU:n vet inte om talet Y är
med eller utan tecken.

Det bestämmer programmeraren.

Dvs, ALU:n gör alltid på samma sätt.

ALU : add, sub

Flaggorna C och V

Bin	2-kompl	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Talcirkeln

add medurs

sub moturs

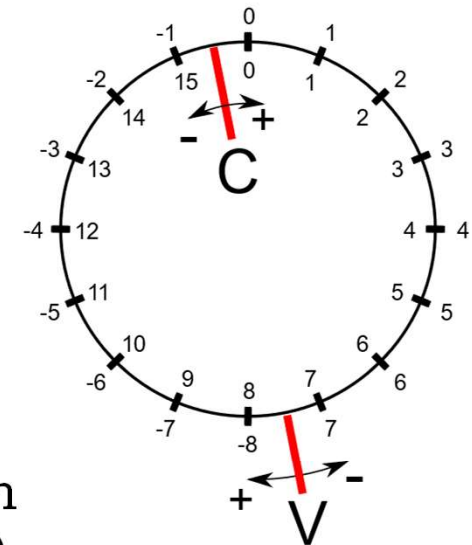
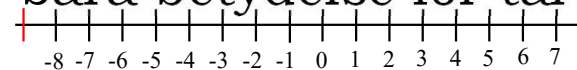
C 1-ställs när man passerar röda linjen från 15→0 (vid add), eller 0→15 (vid sub).

V 1-ställs när man passerar röda linjen från 7→-8 eller -8→7 (vid add med lika tecken eller sub med olika tecken).

C har bara betydelse för tal utan tecken.



V har bara betydelse för tal med tecken.



ALU : add, sub

Flaggorna C och V

Bin	2-kompl	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Exempel

Addition

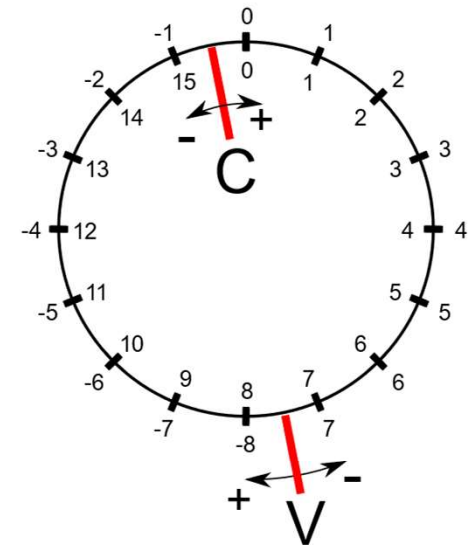
0011 3 3
+1100 -4 12
01111 -1 15
Z:0,N:1,C:0,V:0

0011 3 3
+0101 5 5
01000 -8 8
Z:0,N:1,C:0,V:1

Subtraktion

0011 3 3
-0101 5 5
11110 -2 14
Z:0,N:1,C:1,V:0

1001 -7 9
-0100 4 4
00101 5 5
Z:0,N:0,C:0,V:1



ALU : add, sub

Flaggorna Z, N, C och V i VHDL

	0	1	1	0	A(3 downto 0)	
+	1	1	0	0	B(3 downto 0)	
	1	0	0	1	0	R(4 downto 0)

```
R <= '0' & A + '0' & B;
```

```
Y <= R(3 downto 0);
```

```
...
```

```
Z <= '1' when (R(3 downto 0) = 0) else '0';
```

```
N <= R(3);
```

```
C <= R(4);
```

```
V <= (A(3) and B(3) and not R(3)) or  
      (not A(3) and not B(3) and R(3)) when op=ADD else  
      (not A(3) and B(3) and R(3)) or  
      (A(3) and not B(3) and not R(3)) when op=SUB else  
      '0';
```

ALU : mul, muls, mulsu

Flaggorna Z, N, C, och V

mul (utan tecken)

$$\begin{array}{r} 0011 3 \\ * 1001 9 \\ \hline 0011^{(3)} 1^*0011 \\ 0000 0^*0011 \\ 0000 0^*0011 \\ + 0011 (=24) 8^*0011 \\ \hline 00011011 27 \end{array}$$

muls (med tecken)

$$\begin{array}{r} 0011 +3 \\ * 1001 -7 \\ \hline 0000^{(3)} 0^*0011 \\ 0000 0^*0011 \\ 0000 0^*0011 \\ + 1110 (= -24) -8^*0011 \\ \hline 11101011 -21 \end{array}$$

Algoritmerna blir olika!

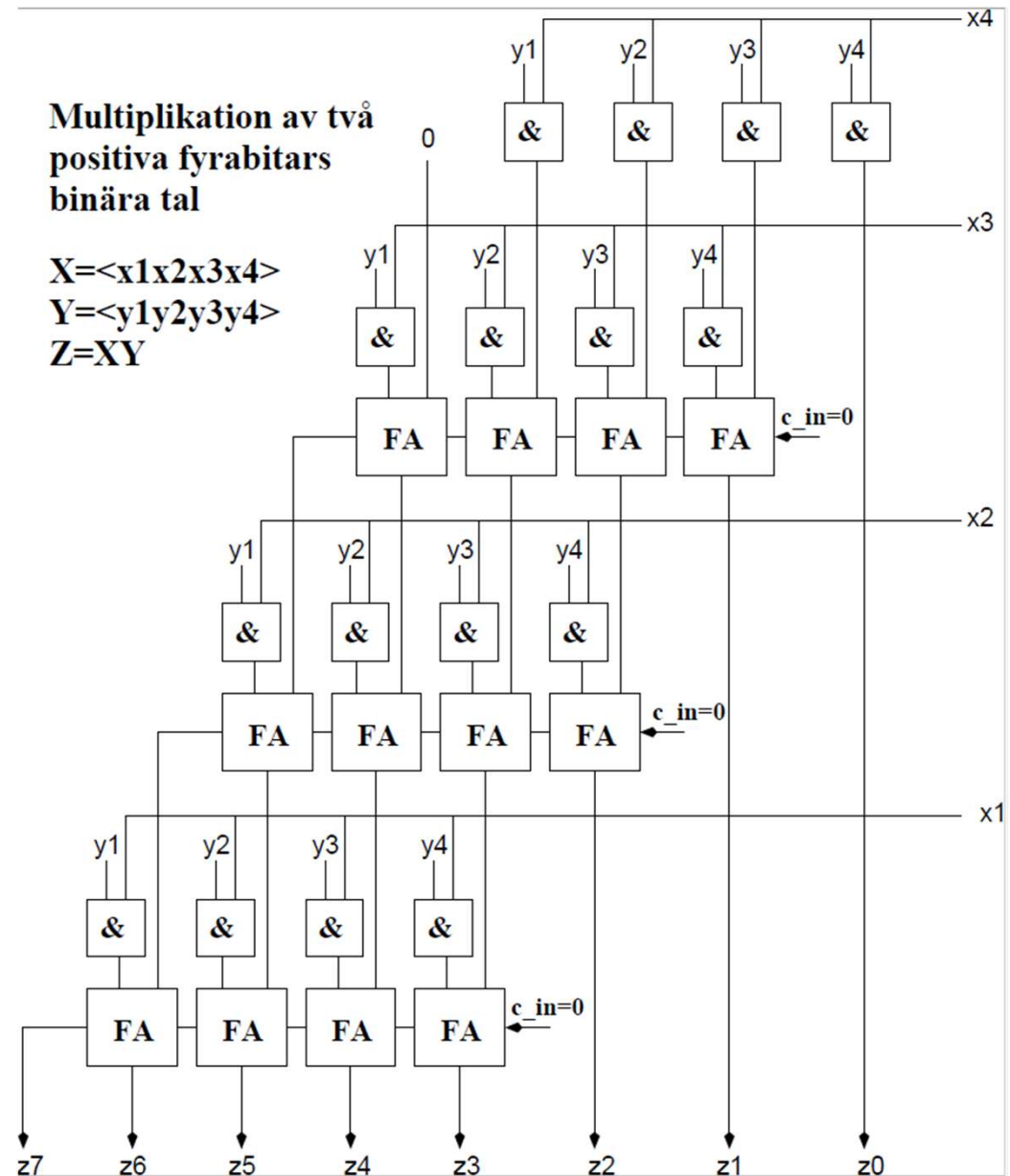


ALU : mul, muls, mulsu

Flaggorna Z, N, C, och V

Hårdvaran för **mul** →

Lång kritisk väg!! →
Men FPGA:n har
inbyggda multi-
plikatorer.



ALU : mul, muls, mulsu

Flaggorna Z, N, C, och V

Flaggorna då?

Z och N, bildas på samma sätt som förut, fast för resultatet med dubbel bredd.

V är opåverkad, dvs multiplikation ger inte overflow.

C sätts vanligen till värdet av MSB

(Most Signifikant Bit), dvs samma som N.

Underlättar teckenutökning i ett program då man vill göra större multiplikationer än vad hårdvaran klarar av. Kräver då instruktioner såsom add, addc, sub och subc.

ALU : div, divs, (divsu)

Flaggorna Z, N, C, och V

Division (heltalsdivision) ger en kvot (Q) och en rest (R).

$$A / B = [R, Q]$$

Det är inte ovanligt att A använder dubbelt så många bitar som B, Q och R.

T ex:

$$35 / 8 = [3, 4]$$

Dvs:

$$00100011 / 1000 = [0011, 0100]$$

8 bitar

4 bitar

4 bitar

4 bitar

Det medför dock att vissa divisioner inte är möjliga.

T ex $35 / 17$ går inte, eftersom 17 kräver 5 bitar.

Och $35 / 2 = [1, 17]$ ger overflow, då 17 kräver 5 bitar.

ALU : div, divs, (divsu)

Flaggorna Z, N, C, och V

Division (heltalsdivision) ger en kvot (Q) och en rest (R).

$$A / B = [R, Q]$$

Man kan också tänka sig att A, B, Q och R använder
Lika många bitar, men kräver då bredare register eller
specialhantering för att dividera med stora tal.

Division med 0 (noll) ger ett sk exception error,
dvs ett undantag som vanligen innebär att ett
särskilt avbrott sker.

ALU : div, divs, (divsu)

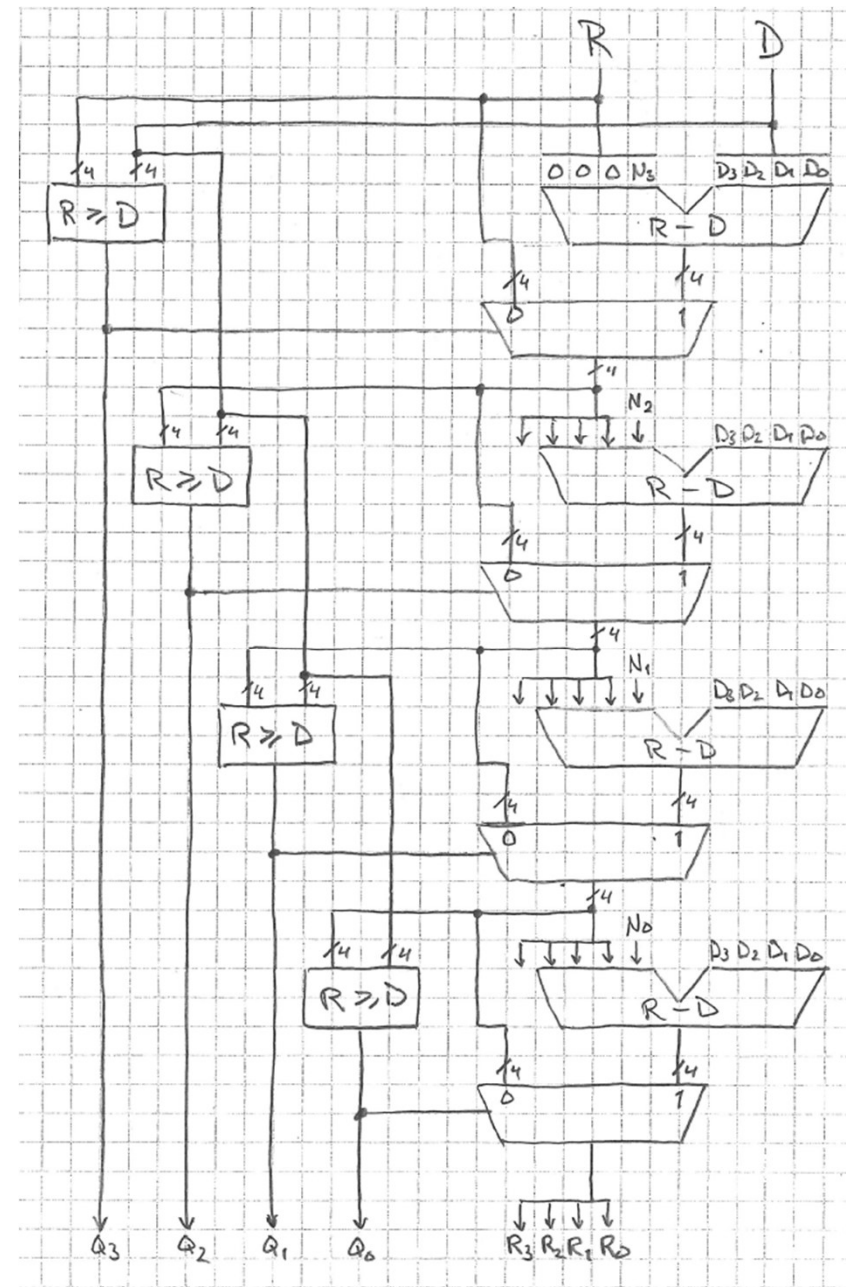
Flaggorna Z, N, C, och V

Hårdvara för **div** →
(utan tecken)

$$\frac{[N_3N_2N_1N_0]}{[D_3D_2D_1D_0]} = [R_3R_2R_1R_0, Q_3Q_2Q_1Q_0]$$

Lång kritisk väg!! →
Dvs, FPGA:n kanske inte
hinner producera resultat
mellan klockflankerna.

/ och mod i VHDL kan alltså
kräva mycket hårdvara!



ALU : div, divs, (divsu)

Flaggorna Z, N, C, och V

Flaggorna då?

Z och N, bildas på samma sätt som tidigare, fast bara för kvoten Q.

V kan inträffa om kvoten Q är större än vad som ryms i målregistret, eller vid division med noll.

C kan "vanligen" nollställas, eller lämnas opåverkad.

Olika processortillverkare gör dock olika med flaggorna.

ALU : VHDL-kod

Bäst är att *dela upp* VHDL-koden för ALU:n i tre delar:

1. Beräkning av resultat (aritmetiskt eller logiskt)
2. Beräkning av flaggor (kandidater)
3. Tilldelning av flaggor (beroende på använd instruktion)

Det blir då *lättare* att skriva koden, *lättare* att testa och felsöka samt *lättare* att utöka funktionaliteten.

ALU : VHDL-kod

1. Beräkning av resultat

```
A : in unsigned(3 downto 0);
B : in unsigned(3 downto 0);
--
signal R : unsigned(7 downto 0);

process(A, B, op)
Begin
  R <= (others => '0'); -- default value
  case op is
    when "000" => R <= (others => '0');           -- No op
    when "001" => R(4 downto 0) <= ('0'&A) + ('0'&B); -- A+B
    when "010" => R(4 downto 0) <= ('0'&A) - ('0'&B); -- A-B
    when "011" => R <= A * B;                       -- unsigned mul
    when "100" => R <= unsigned(signed(A) * signed(B)) -- signed muls
    when "101" => R <= (A rem B) & (A / B);           -- unsigned div
    when "110" => R(7 downto 4) <=                  -- signed divs
      unsigned(signed(A) rem signed(B));
      R(3 downto 0) <=
        unsigned(signed(A) / signed(B));
    when others => null;
  end case;
end process;

Y <= R(3 downto 0); -- låga bitarna av resultatet
H <= R(7 downto 4); -- höga bitarna av resultatet (vid mul och div)
```


ALU : VHDL-kod

2. Beräkning av flaggor

```
-- Zc, Nc, Cc, Vc are candidates for flags
signal Zc, Nc, Cc, Vc : std_logic;

Zc <= '1' when R(7 downto 0)=0 and ((op="011") or (op="100")) else -- mul or muls
      '1' when R(3 downto 0)=0 and (op/"011") and (op/"100") else -- not mul or muls
      '0';

Nc <= R(7) when ((op="011") or (op="100")) else -- mul or muls
      R(3);

Cc <= R(7) when ((op="011") or (op="100")) else -- mul or muls
      R(4);

Vc <= (not A(3) and not B(3) and R(3)) or -- when ...
      (A(3) and B(3) and not R(3)) when (op="001") else -- ... add
      (not A(3) and B(3) and R(3)) or -- when ...
      (A(3) and not B(3) and not R(3)) when (op="010") else -- .. sub
      '0';
```

ALU : VHDL-kod

3. Tilldelning av flaggor

```
-- Z, N, C, V are the actual flags
signal Z, N, C, V : std_logic;

process(clk)
begin
    if rising_edge(clk) then
        if (rst='1') then
            Z <= '0'; N <= '0'; C <= '0', V <= '0';
        else
            case op is
                when "000" => null;
                when "001" => Z<=Zc; N<=Nc; C<=Cc; V<=Vc;           -- add
                when "010" => Z<=Zc; N<=Nc; C<=Cc; V<=Vc;           -- sub
                when "011" => Z<=Zc; N<=Nc; C<=Cc;                   -- mul
                when "100" => Z<=Zc; N<=Nc; C<=Cc;                   -- muls
                when "101" => Z<=Zc; N<=Nc; C<='0';                 -- div
                when "110" => Z<=Zc; N<=Nc; C<='0';                 -- divs
                when others => null;
            end case;
        end if;
    end if;
end process;
```

För övriga operationer, titta t ex på databladet för AVR-processorn.

ALU : VHDL-kod

Beräkningar med större resultat än vad ALU:n klarar

Antag en 4-bitars ALU, men att vi vill addera två 8-bitars tal

```
  11  1  1  1
  0110 1101 = $6D = 109
+0010 0101 = $25 =  37
-----
 1001 0010 = $92 = 146
```

Pseudokod "8-bitars addition"

ADD \$D,\$5 ; \$D+\$5 = \$12

ADC \$6,\$2 ; \$6+\$2+\$1 = \$09

VHDL-kod i ALU:n

```
...
when "001" => R(4 downto 0) <= ('0'&A) + ('0'&B);           -- ADD
when "001" => R(4 downto 0) <= ('0'&A) + ('0'&B) + ("0000"&C); -- ADC
when "010" => R(4 downto 0) <= ('0'&A) - ('0'&B);           -- SUB
when "010" => R(4 downto 0) <= ('0'&A) - ('0'&B) - ("0000"&C); -- SBC
...
```

Man kan använda bara ADC men måste då alltid nollställa Carry innan ALLA additioner (även bara 4-bitars) för att resultatet säkert ska bli korrekt.

Pseudokod "8-bitars addition" med bara ADC

CLC ; Clear carry

ADC \$D,\$5 ; \$D+\$5 = \$12

ADC \$6,\$2 ; \$6+\$2+\$1 = \$09

ALU : Fixtal

Hur representerar datorn tal med decimaler?

Vad händer när ett tal shiftas åt höger (delas med 2):



Det blir en "decimalpunkt" mellan registret och carry-biten, men i princip kan decimalpunkten placeras var som helst, bara den har en fix position i hela talsystemet. T ex:

$$\begin{array}{ccccccc} 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\ \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \end{array} = 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4} = 11.6875$$

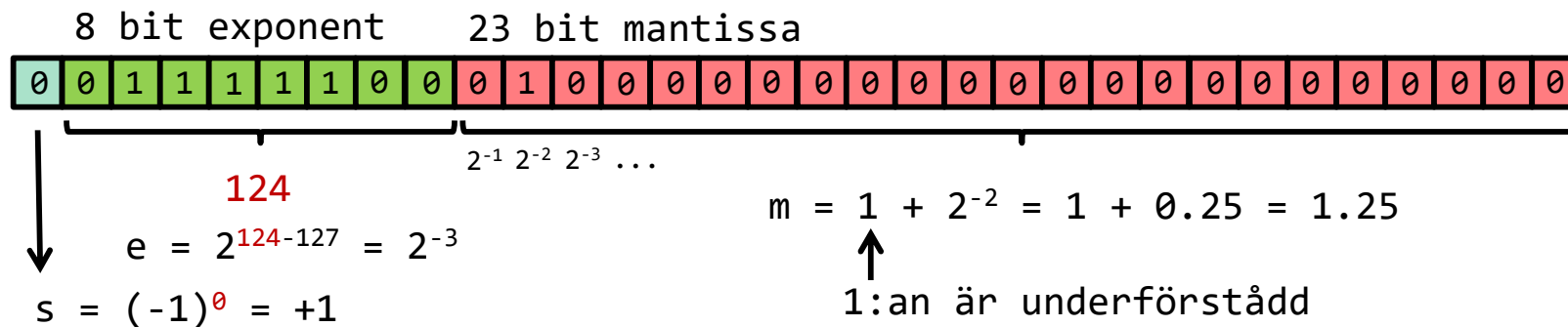
Det är då ett fixtal och beräkningar kan göras precis som tidigare. Enbart tolkningen gör det till ett fixtal.

Dvs, vid utskrift (el inmatning) tolkas talet enligt positionen för decimalpunkten.

ALU : Flyttal

Hos ett flyttal hamnar decimalpunkten på olika platser beroende på talets storlek. Dvs, decimalpunkten *flyter omkring*.

Ett decimaltal, t ex $0.15625 = 1.5625 \cdot 10^{-1}$, och 1.5625 kallas mantissa och -1 kallas exponent. Talet kan lagras med basen 2 enligt $\text{mantissa} \cdot 2^{\text{exponent}}$, men då med andra värden på mantissa och exponent (anpassade till basen 2), och enligt en viss standard. T ex för ett 32-bitars flyttal:



$$\text{Värdet : } s * m * e = (+1) * 1.25 * 2^{-3} = 0.15625$$

ALU : Flyttal

Hur adderar man två flyttal?

Antag: $3.5 + 0.15625$

$$3.5 = 1.75 * 2^1 \text{ (lagrat som flyttal)}$$

$$0.15625 = 1.25 * 2^{-3} \text{ (lagrat som flyttal)}$$

För att kunna addera talen måste båda ha samma exponent.

Vi skriver om (skiftar höger 4 steg) det mindre talet:

$$0.15625 = 1.25 * 2^{-3} = 0.078125 * 2^1$$

$$1.75 + 0.078125 = 1.828125 \Rightarrow$$

$$3.5 + 0.15625 = 1.828125 * 2^1 = 3.65625$$

Vi kan alltså inte utan vidare ta fixtalen och skicka in i vår ALU. Mantissor och exponenter måste hanteras på rätt sätt.

Fixtal och Flyttal, BCD-tal

Det går inte att representera vissa decimaltal exakt med fixtal eller flyttal, oavsett hur många bitar talet lagras med.

T ex:

$$0.1 \approx 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} = 0.099609375 = 0.000110011_2$$

Ju fler bitar ju bättre precision, men aldrig exakt.

Så hur kan man göra då? Om man vill ha full precision?

BCD-aritmetik!

Dvs, lagra siffrorna en och en i var sitt 4-bitars tal.

Använd en "vanlig" ALU och addera siffra för siffra med hjälp av ett program eller en algoritm.

Det går förstås att bygga en special-ALU, alltså en BCD-ALU, men den får då ganska begränsad användning.

Datorkonstruktion TSEA83

Adressavkodning

IO-manager

Subrutin

Avbrott

Hierakisk konstruktion

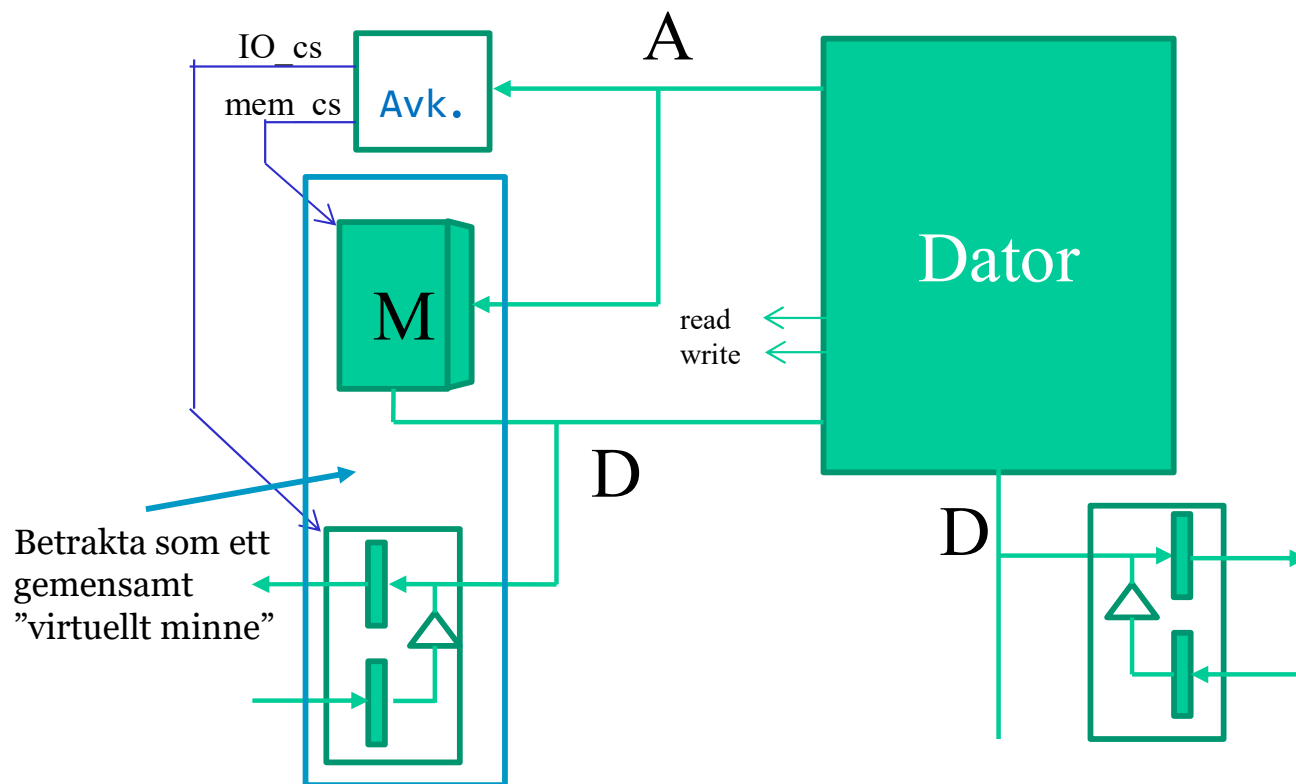
Boot-loader

Timing

Arbetsgång

Tidrapport

Adressavkodning eller I/O-mappning?



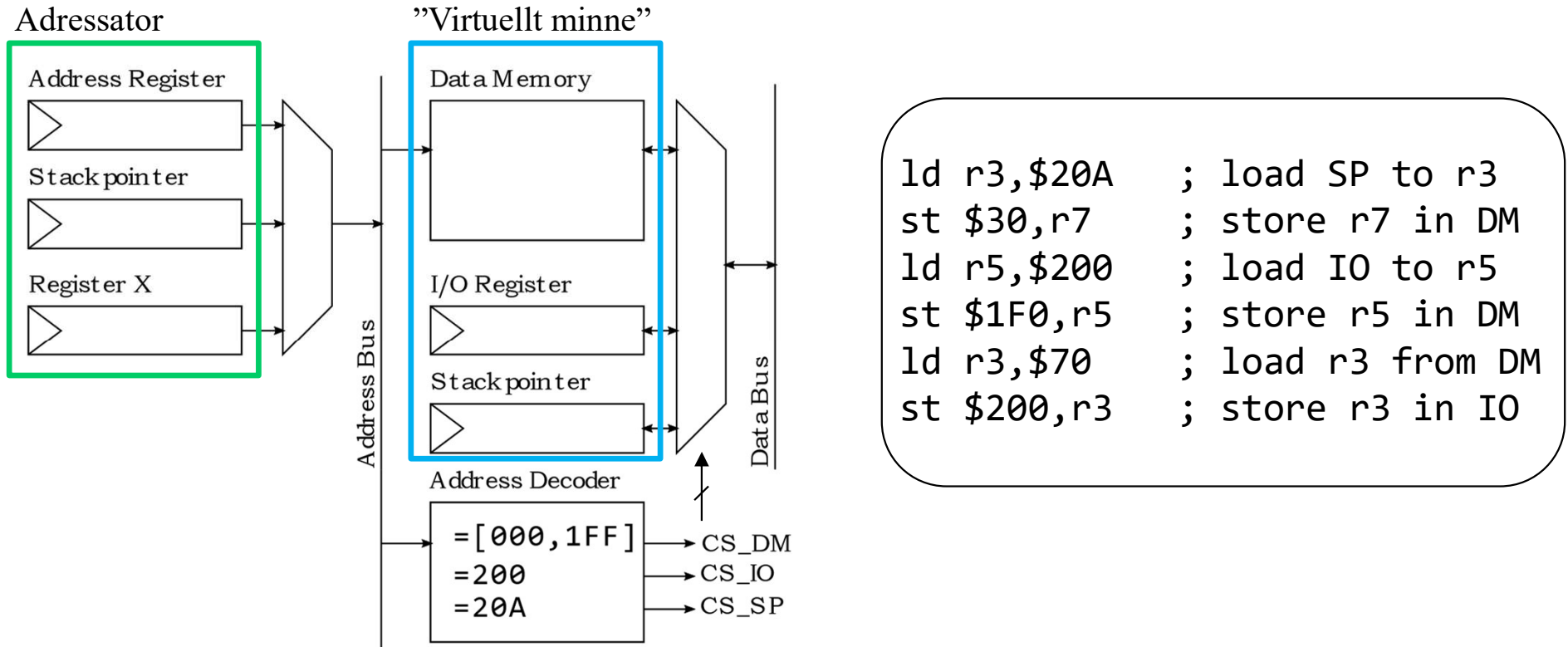
Memory-mapped I/O

- På den yttre databussen
- **I/O-registren som minnesceller**
- Vanliga instruktioner (LDA, STA)
- Adressavkodning måste fixas mm

I/O-mapped I/O

- Direkt på interna databussen
- **I/O-registren som interna register**
- Speciella instruktioner,
- Som kan vänta på att någonting hänt (fördelen med mikroprog. dator)

Adressavkodning



Adressbussen får sitt värde från något register, t ex adress-register, stackpekare eller nåt annat register.

Adress-avkodaren skapar chip-select-signaler, beroende på adress-bussens värde.

Chip-select-signalerna avgör vilket register / vilken enhet som skriver till eller läser från databussen.

Adressavkodning

```
signal CS_DM, CS_IO, CS_SP : std_logic;

signal DATA_BUS : unsigned(7 downto 0);

signal ADR_BUS : unsigned(15 downto 0);

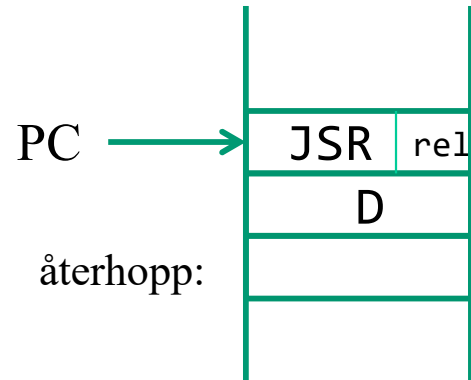
CS_DM <= '1' when (ADR_BUS >= 0) and (ADR_BUS <= x1FF) else '0';
CS_IO <= '1' when (ADR_BUS = x200) else '0';
CS_SP <= '1' when (ADR_BUS = x20A) else '0';

DATA_BUS <= DM(ADR_BUS) when (CS_DM = '1') and (READ = '1') else
            IO_REG when (CS_IO = '1') and (READ = '1') else
            SP_REG when (CS_SP = '1') and (READ = '1') else
            ...

process(clk)
begin
    if rising_edge(clk) then
        if (rst='1') then
            SP_REG <= (others => '0');
        elsif (CS_SP = '1') and (WRITE = '1') then
            SP_REG <= DATA_BUS;
        end if;
    end if;
end process;
```

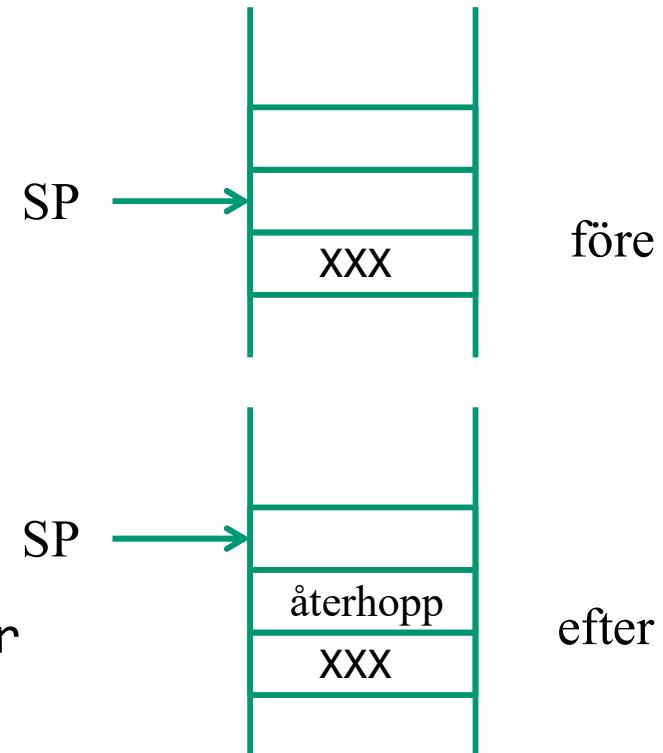
Subrutinhopp JSR D

Subrutinhopp JSR D



JSR D

- Självrelativ a-mod
beräkna hoppadress $PC+2+D \rightarrow tr$
- exe
 $PC+2 \rightarrow mem(SP)$, $SP--$
 $tr \rightarrow PC$



Subrutinåterhopp RTS



* underförstådd a-mod

* exe

```

    SP++                ; justera SP
    M(SP) -> PC        ; hoppa tillbaka
    
```

Avbrott på mikroprog. CPU

Att implementera en avbrottsmekanism för en avbrottskälla är relativt enkelt. Se tidigare föreläsning.

Princip

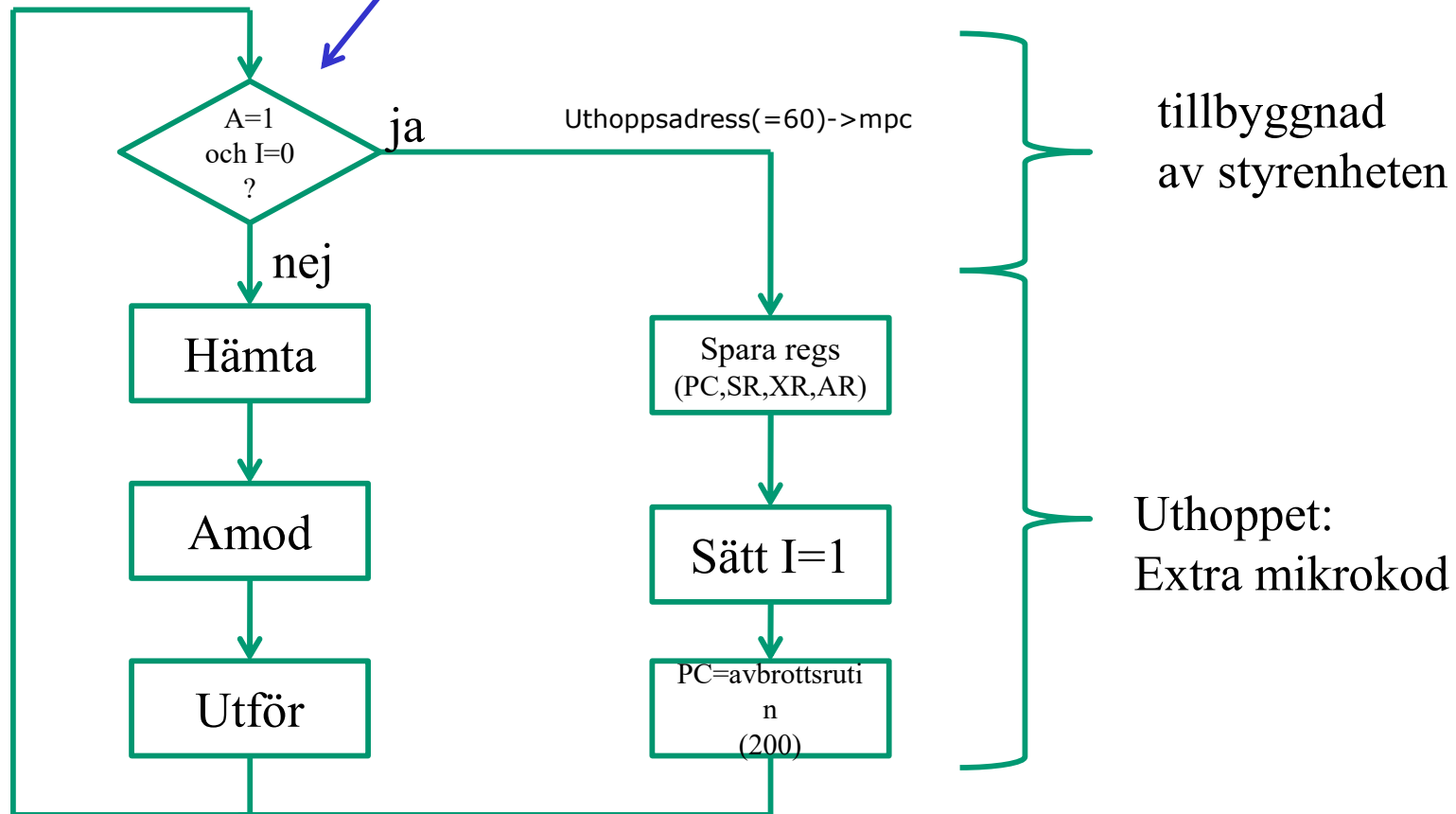
- Kontrollera om avbrott kan utföras (spärrvippan = 0)
- Sätt spärrvippan=1 (avbryt inte pågående avbrott)
- Kör färdigt pågående instruktion
- Hoppa till avbrottsvektor
- Kör avbrottsrutin, ISR.
- Återhopp från avbrott
- Återställ spärrvippan=0

Flera avbrottskällor kräver mer funktionalitet.

- Ska avbrott prioriteras, dvs att avbrott kan avbryta varandra?
- Hur noterar man andra avbrott under ett pågående avbrott?

Avbrott

Styrenheten kollar om det är avbrott mellan två instruktioner



Avbrott på pipeline-CPU

Svårare än på en mikroprogrammerad CPU då hänsyn måste tas till vad som pågår i pipen.

”Lättaste” metoden: töm pipen (dvs kör klart pågående instruktioner) och kör sedan avbrottet. Dock något ineffektivt ...

Smartare? Injisera en ”avbrottsinstruktion” i pipen (på lämplig plats), p s s som NOP injiseras vid ett hopp, samt betrakta avbrottet lite som ett subrutinanrop.

Strax innan avbrott...

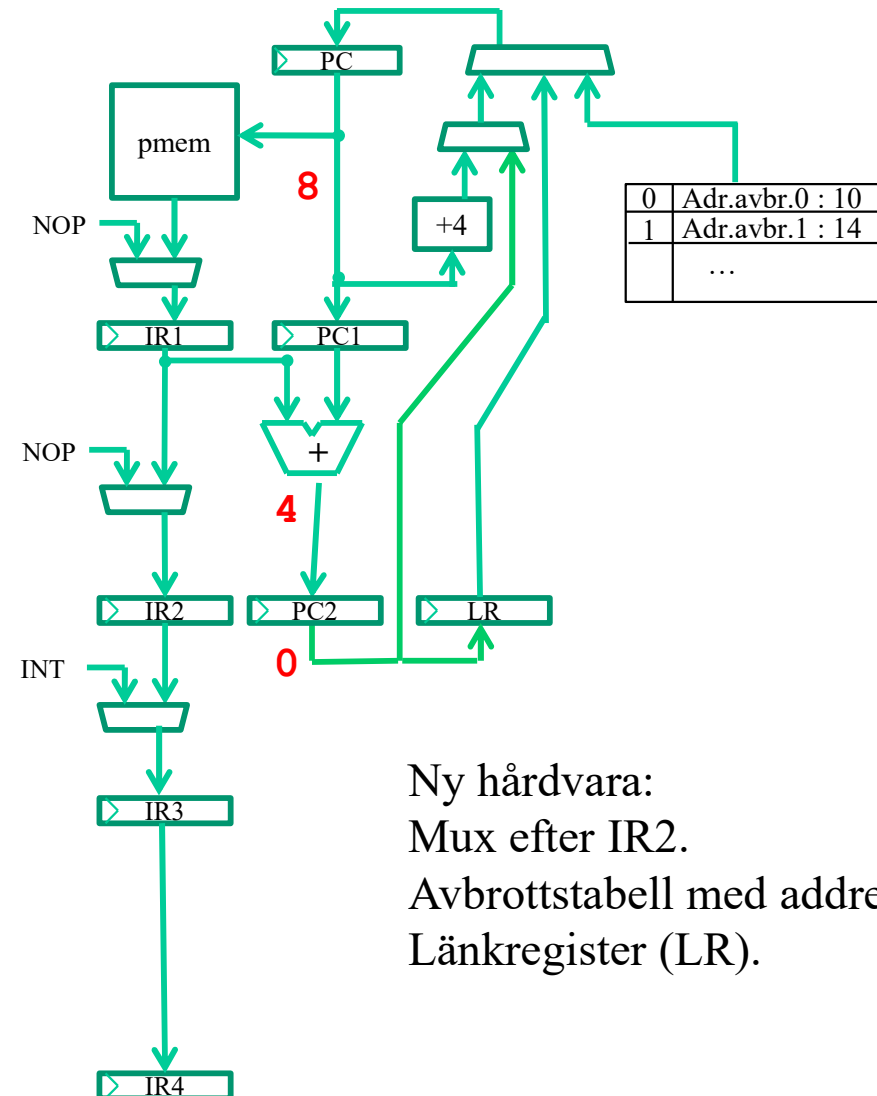
0:XXX
4:YYY
8:ZZZ
C:WWW

Avbrott 1
14:....
18:....
1C:RETI
20:AAA
24:BBB

ZZZ

YYY

XXX



Ny hårdvara:
Mux efter IR2.
Avbrottstabell med adresser.
Länkregister (LR).

Avbrott: Injiserat INT

0:XXX
4:YYY
8:ZZZ
C:WWW

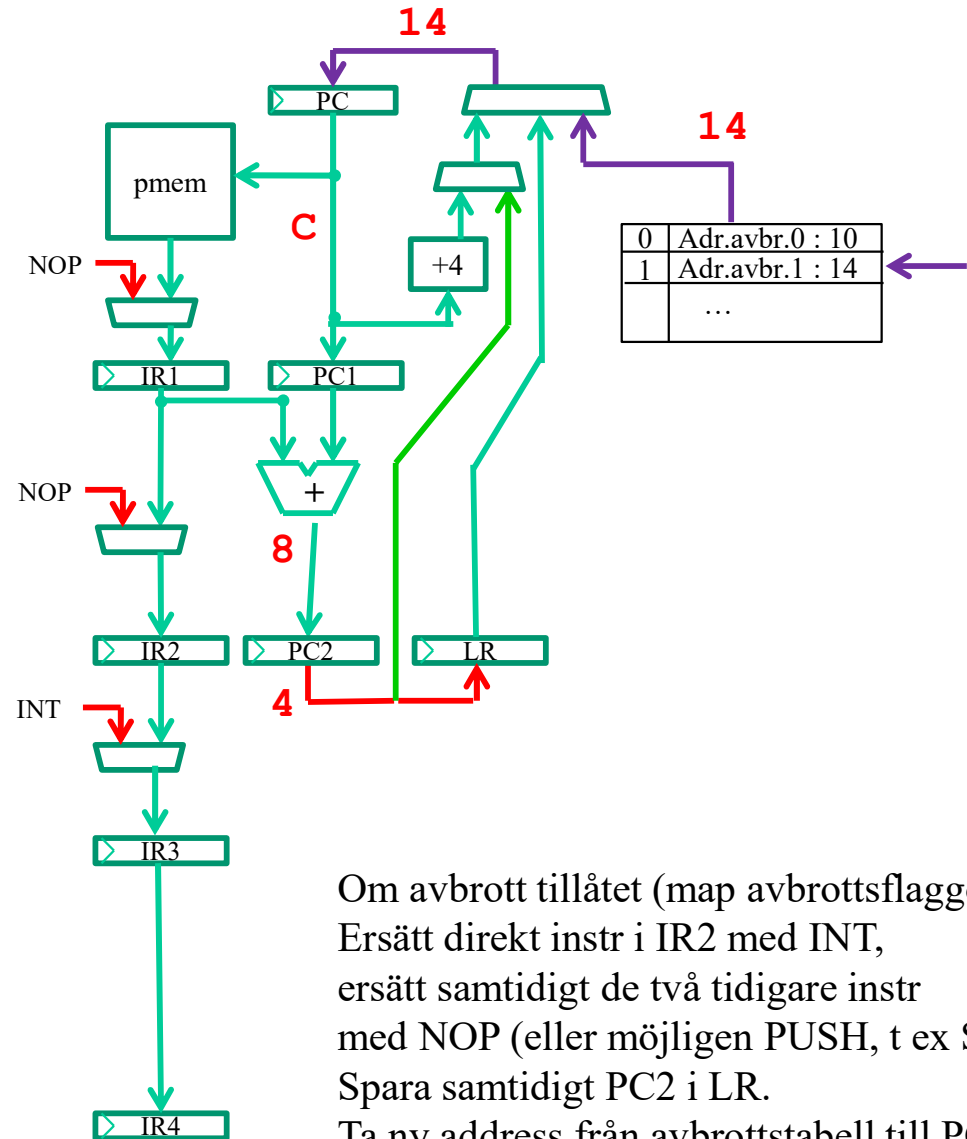
Avbrott 1
14:....
18:....
1C:RETI
20:AAA
24:BBB

~~WWW~~ NOP

~~ZZZ~~ NOP

~~YYY~~ INT

XXX



Om avbrott tillåtet (map avbrottsflaggor):
Ersätt direkt instr i IR2 med INT,
ersätt samtidigt de två tidigare instr
med NOP (eller möjligen PUSH, t ex SR)
Spara samtidigt PC2 i LR.
Ta ny address från avbrottstabell till PC.
Utöver det, uppdatera avbrottsflaggor,
disabla avbrott.

Återhopp från avbrott

0:XXX
4:YYY
8:ZZZ
C:WWW

Avbrott 1

14:...

18:...

1C:RETI

20:AAA

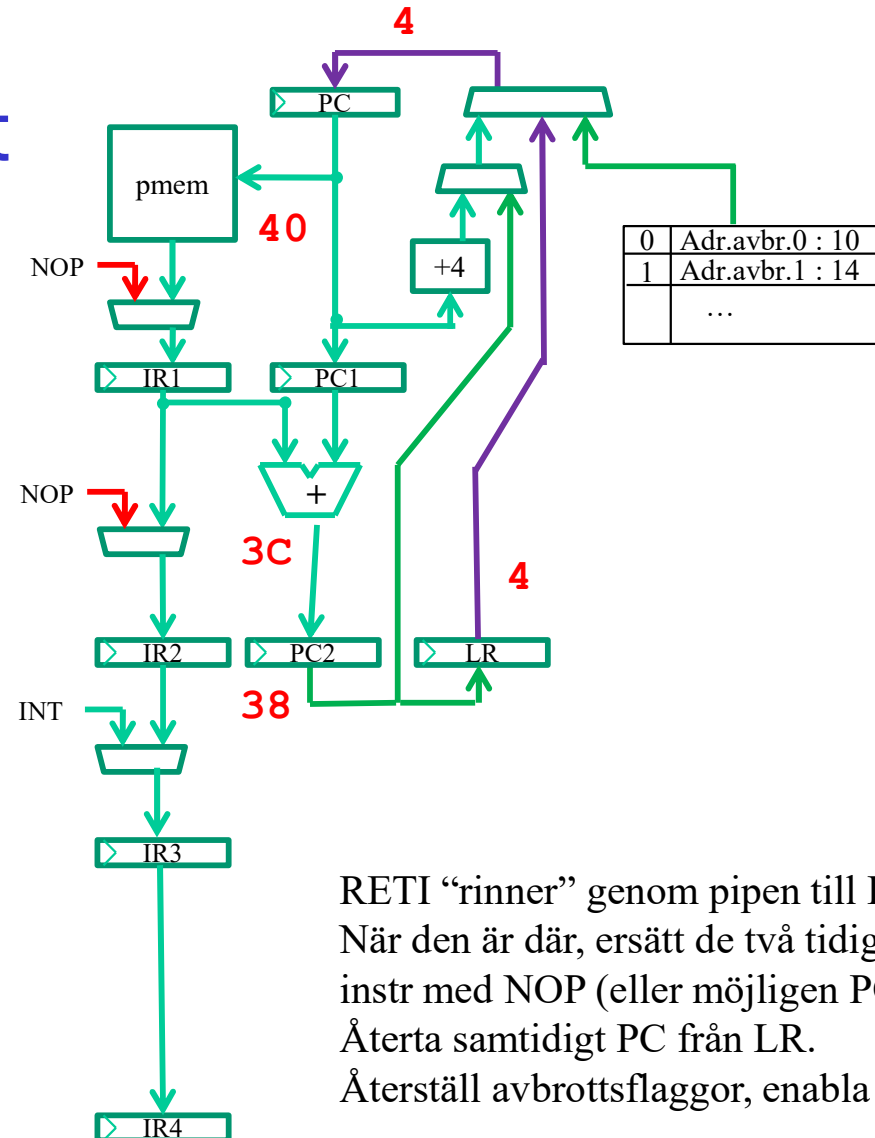
24:BBB

~~BBB~~ NOP

~~AAA~~ NOP

RETI

...

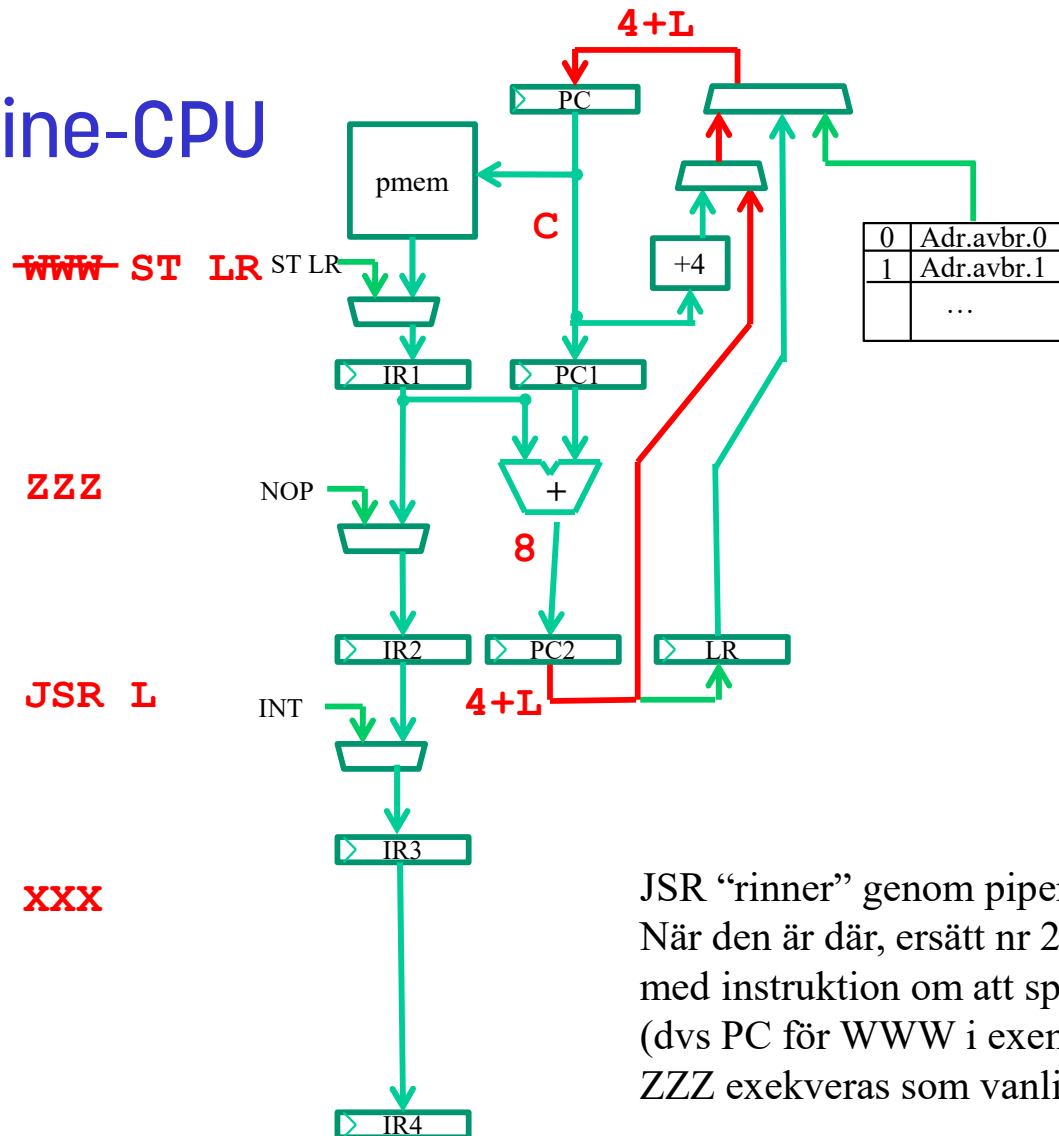


RETI “rinner” genom pipen till IR2.
När den är där, ersätt de två tidigare instr med NOP (eller möjligen POP, t ex SR)
Återta samtidigt PC från LR.
Återställ avbrottsflaggor, enabla avbrott.

Subrutin på pipeline-CPU

0:xxx
4:JSR L
8:ZZZ
C:WWW

Subrutin
14:AAA
18:BBB
1C:RET
20:CCC
24:DDD



JSR “rinner” genom pipen till IR2. När den är där, ersätt nr 2 efter JSR med instruktion om att spara PC i LR (dvs PC för WWW i exemplet). ZZZ exekveras som vanligt, vid ett hopp.

Återhopp från subrutin

0:xxx
4:JSR L
8:ZZZ
C:WWW

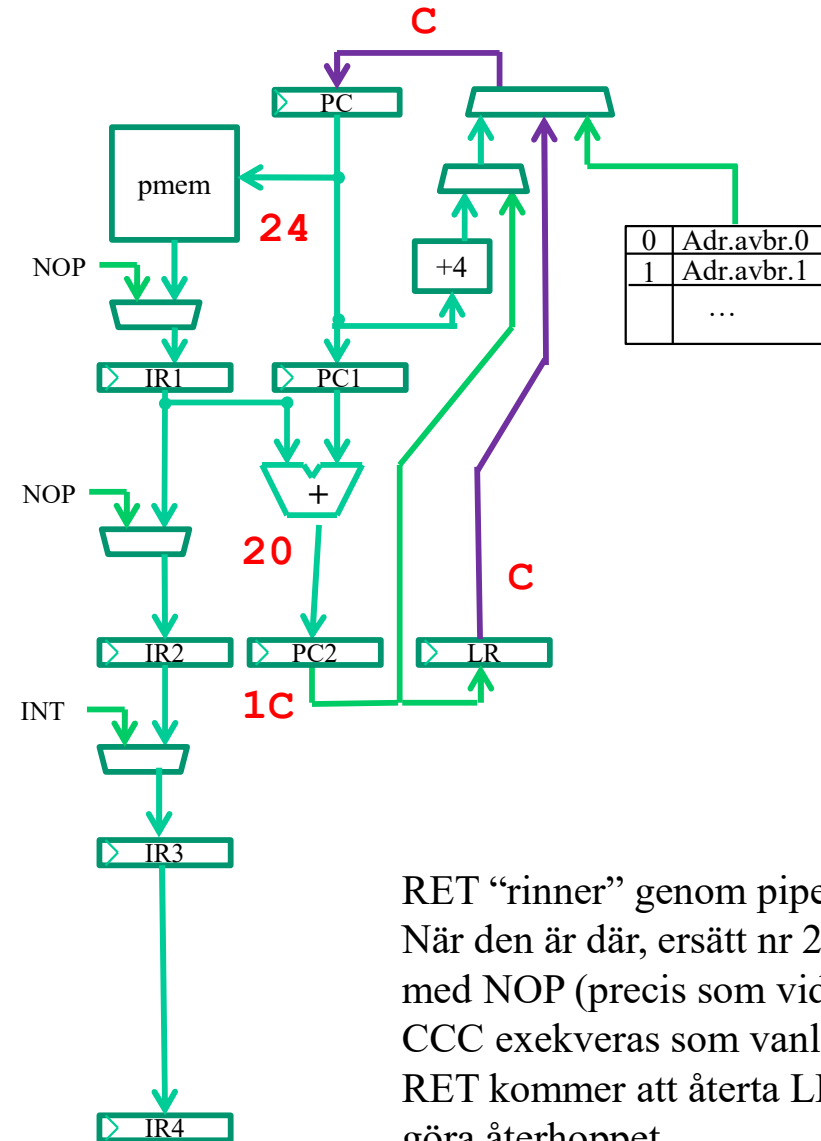
Subrutin
14:AAA
18:BBB
1C:RET
20:CCC
24:DDD

~~DDD~~ NOP

CCC

RET

BBB



Avbrott och subrutin på pipelinad CPU

Tekniken med ett länk-register (LR) är vanligt förekommande i pipelinade processorer. Det är relativt enkelt att kopiera återhoppss-adressen till ett register (LR) för att sedan återta den vid behov.

Nackdelen är att hårdvaran bara erbjuder en nivå av avbrott / subrutiner då man bara kan hålla en adress år gången i LR.

För att klara flera nivåer av anrop får programmeraren spara/hämta LR på stacken med push/pop.

LR kan av det skälet lämpligen utgöras av ett av de generella registren i registerfilen, men då med lite speciella datavägar.

Avbrott/subrutin

```
push {LR}
```

```
... ev. anrop till subrutin
```

```
...
```

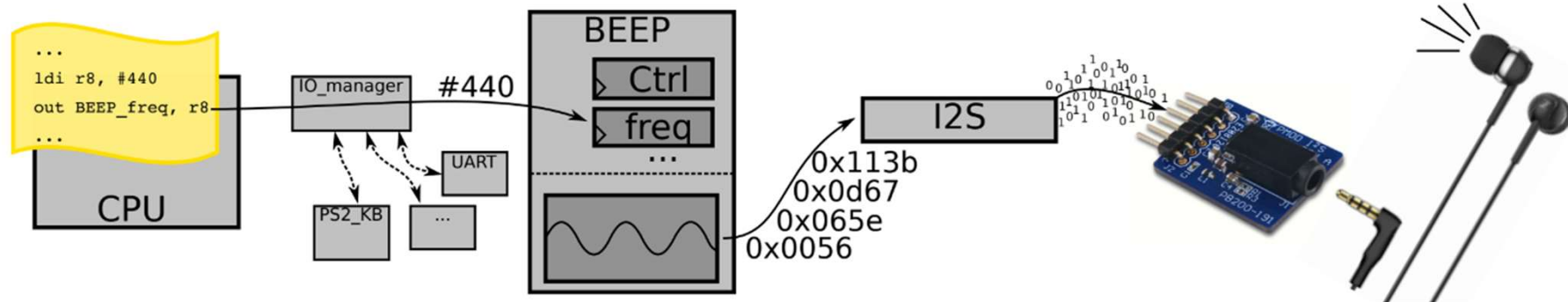
```
pop {LR}
```

```
reti/ret
```

IO-manager

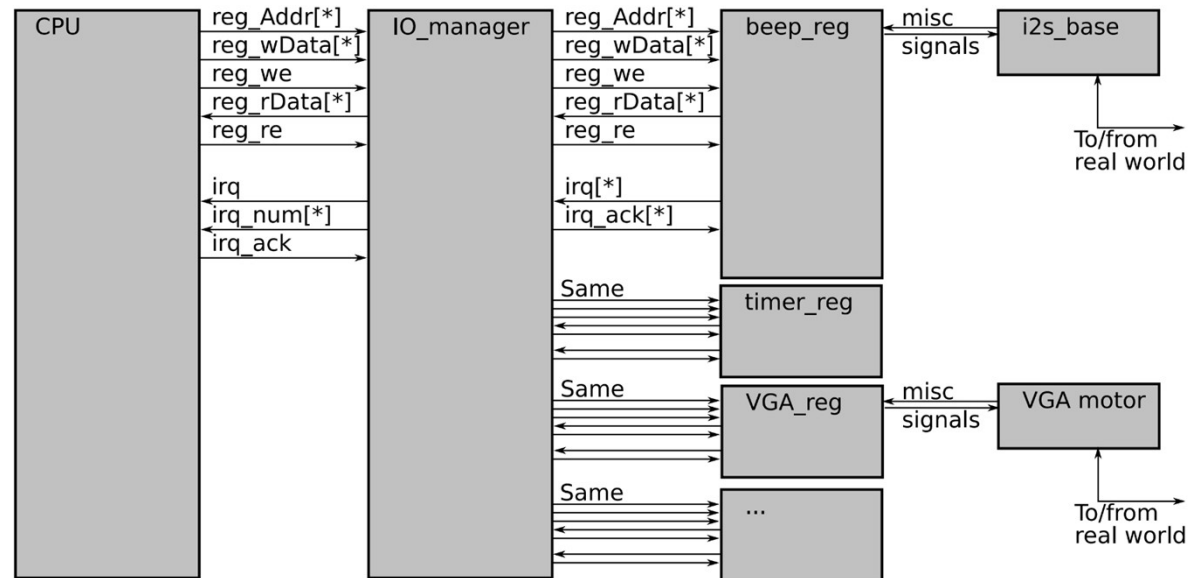
Vi erbjuder, på försök, en s k IO-manager.

Den ger en möjlighet att ansluta IO-enheter och avbrottskällor på ett konsekvent sätt

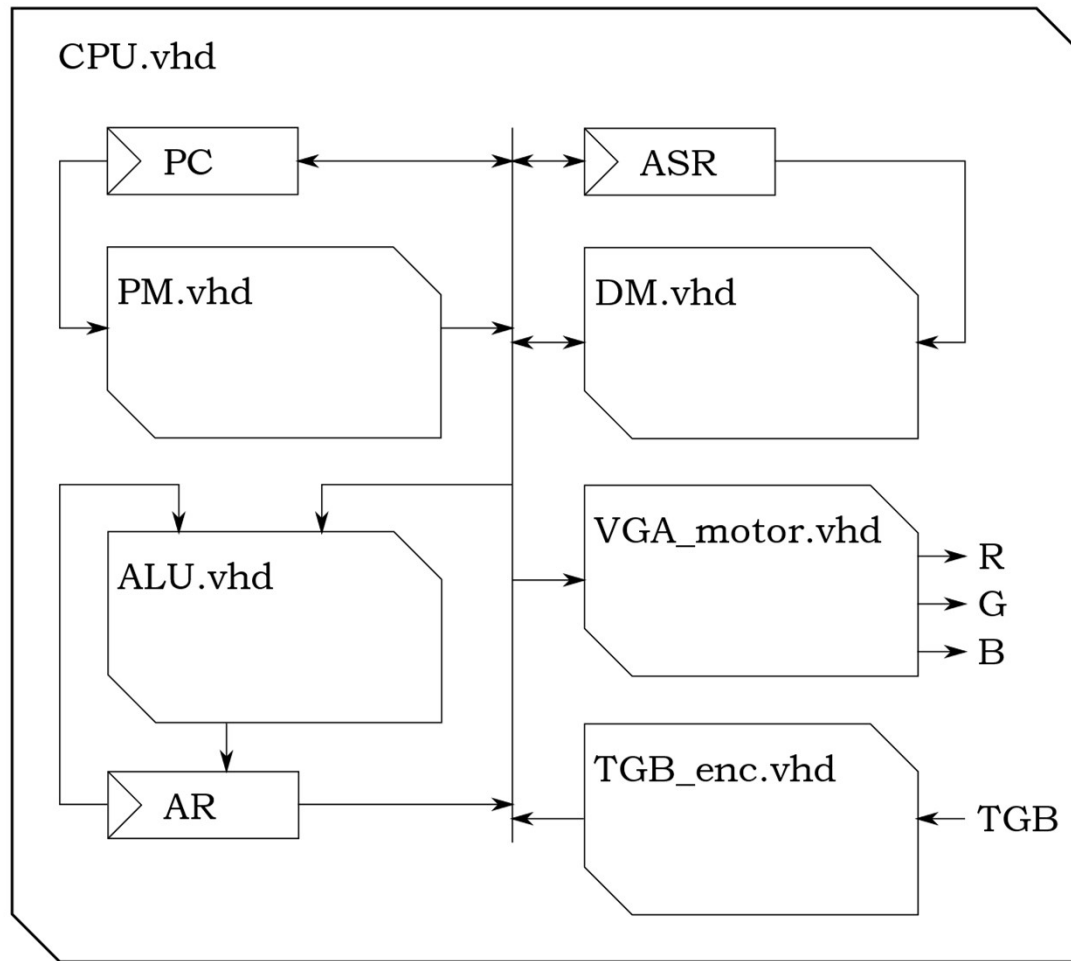


IO-managern behöver dock anpassas till den egna situationen och lönar sig främst om man har ett flertal externa enheter.

Läs "datablad" på kurshemsidan.



Hierakisk / modulbaserad konstruktion och katalogstruktur



Katalogstruktur

```
/proj/CPU.vhd  
/proj/CPU_tb.vhd  
/proj/Makefile  
/proj/Basys3.xdc  
/proj/ALU/ALU.vhd  
/proj/ALU/ALU_tb.vhd  
/proj/ALU/Makefile  
/proj/DM/DM.vhd  
/proj/DM/DM_tb.vhd  
/proj/DM/Makefile
```

Versionshantera koden, med t ex Git via gitlab.liu.se

Hierakisk / modulbaserad konstruktion och katalogstruktur

```
entity BLOCK_RAM is
  port ( clk          : in std_logic;
        -- port 1
        we1           : in std_logic;
        data_in1       : in std_logic_vector(7 downto 0);
        data_out1      : out std_logic_vector(7 downto 0);
        addr1          : in unsigned(10 downto 0);
        -- port 2
        we2           : in std_logic;
        data_in2       : in std_logic_vector(7 downto 0);
        data_out2      : out std_logic_vector(7 downto 0);
        addr2          : in unsigned(10 downto 0));
end BLOCK_RAM;

-- Block RAM type
type ram_t is array (0 to 2047) of std_logic_vector(7 downto 0);
-- initiate Block RAM
Shared variable BRAM : ram_t := (others => (others => '0'));

process(clk)
begin
  if rising_edge(clk) then
    if (we1 = '1') then
      BRAM(to_integer(addr1)) := data_in1;
    end if;
    data_out1 <= BRAM(to_integer(addr1));
  end if;
end process;

process(clk)
begin
  if rising_edge(clk) then
    if (we2 = '1') then
      BRAM(to_integer(addr2)) := data_in2;
    end if;
    data_out2 <= BRAM(to_integer(addr2));
  end if;
end process;
```

Med t ex Block-RAM som en modul (komponent) så går det bara att koppla in den övriga konstruktionen via interfacet (portdeklarationen). Det är **BRA!** (enl kod till vänster)

I annat fall, dvs om Block-RAM:et inte är en modul, så går det att göra accesser till Block-RAM:et lite överallt i koden. Det är **DÅLIGT!** (enl kod nedan)

T ex:

```
data1 <= BRAM(addr1);
data2 <= BRAM(addr2);
data3 <= BRAM(addr3);
```

Dvs, tre samtidiga accesser från olika adresser. Syntesverktyget tvingas lösa det genom att göra kopior av Block-RAM:et, vilket kanske gör att konstruktionen inte får plats.

Boot-loader

Mot slutet av projektet, när konstruktionen blir större, kommer det ta längre och längre tid att syntetisera. Samtidigt handlar det mesta arbetet då om att skriva programmet till datorn.

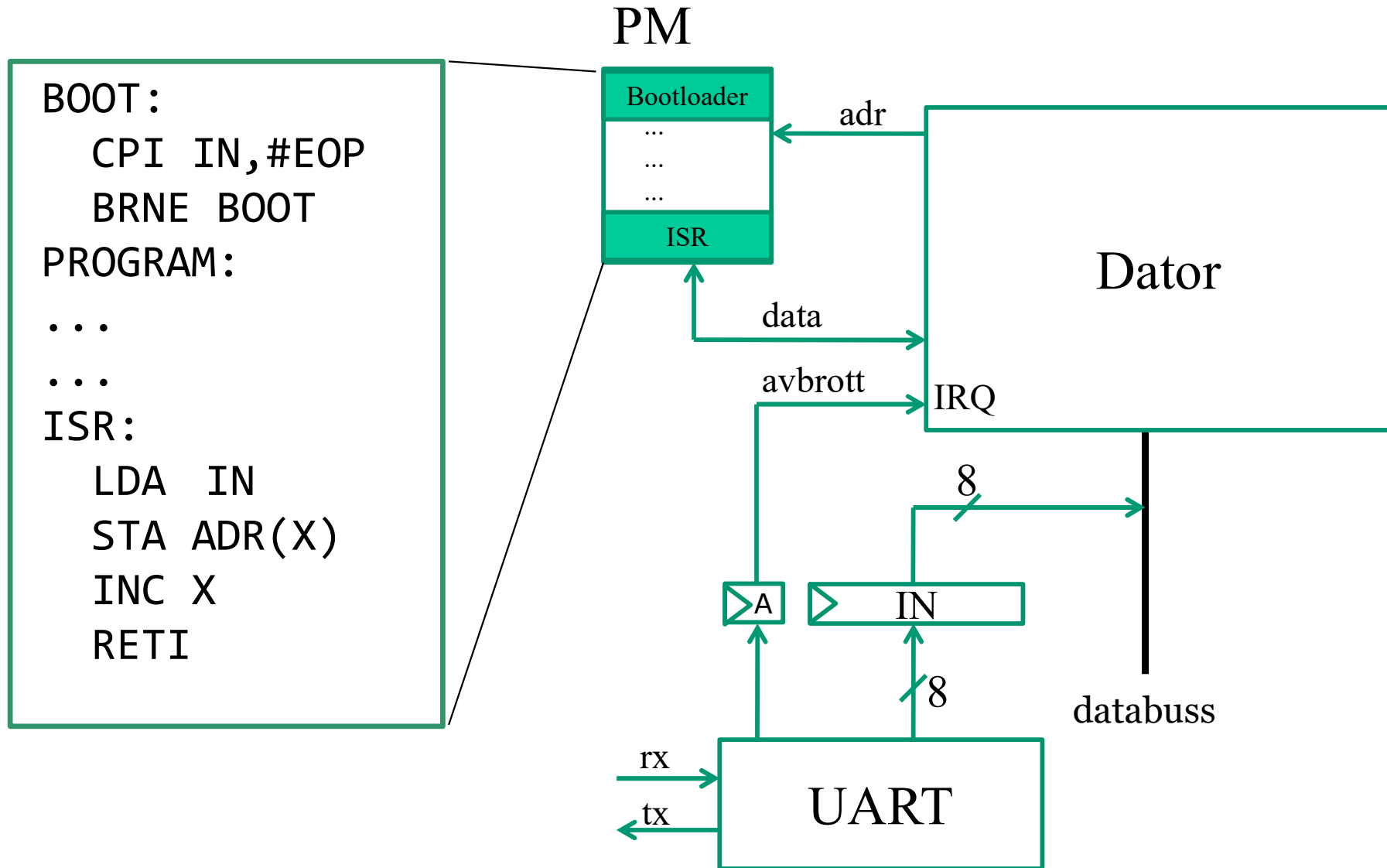
Dvs, varje liten programförändring kan då ta flera minuter att testa, om programkoden skrivs direkt i programminnet i VHDL-koden.

Det kan löna sig att bygga en boot-loader vars enda uppgift är att vänta på inkommande data/programkod från en UART, placera in det på rätt plats i minnet på den körande datorn och sedan starta programmet.

Dvs, det enda program som finns i datorn från början är boot-loader-programmet.

Boot-loader, alt 1

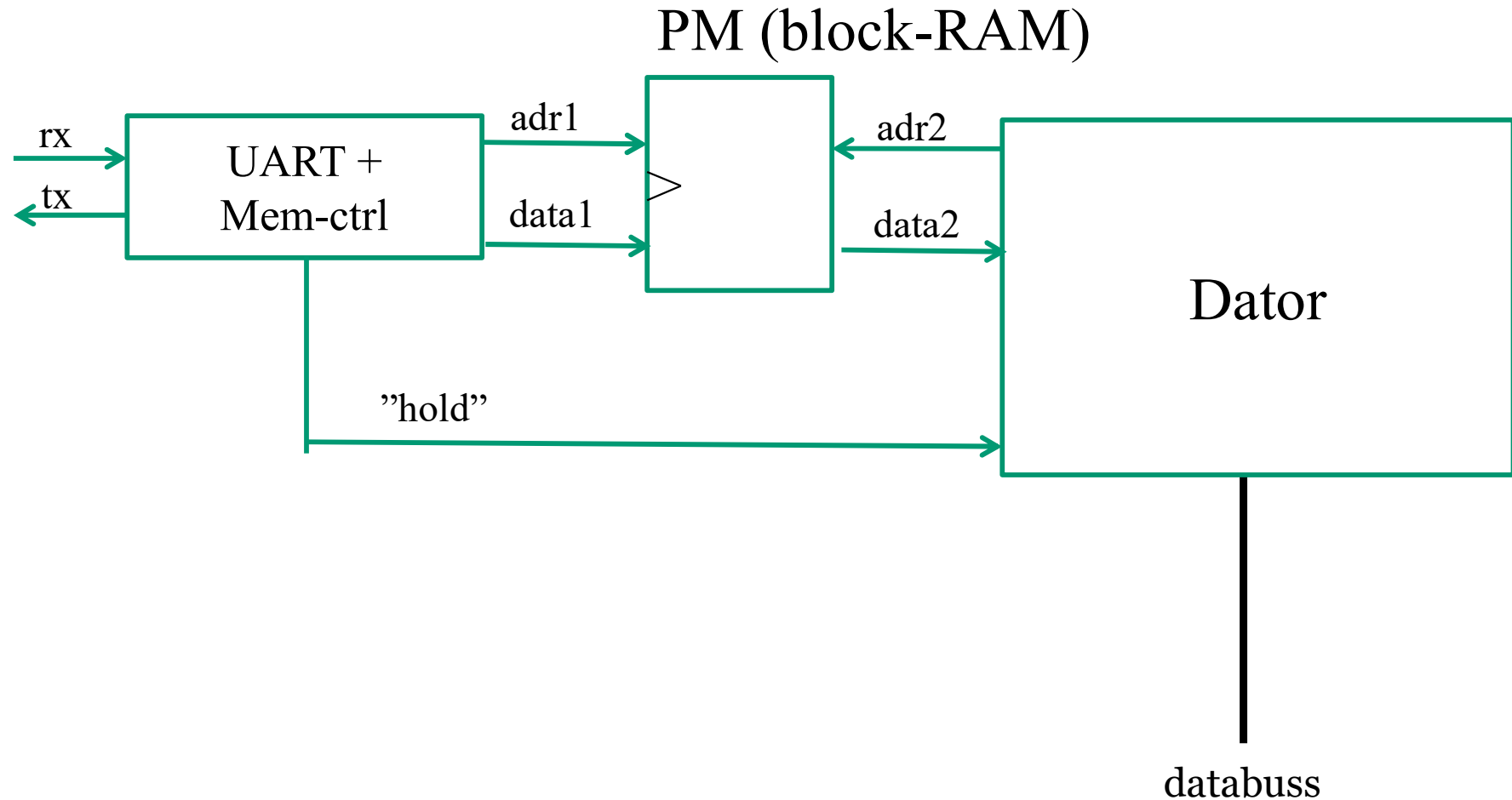
Programkod kan laddas från en extern dator via UART, in i den körande datorn och skrivs in i programminnet via ett boot-loader-program.



EOP = kod för End Of Program

Boot-loader, alt 2

Bygg programminnet PM som ett block-RAM. Skriv direkt till minnet (DMA) och håll datorn stilla tills programmet laddats in.

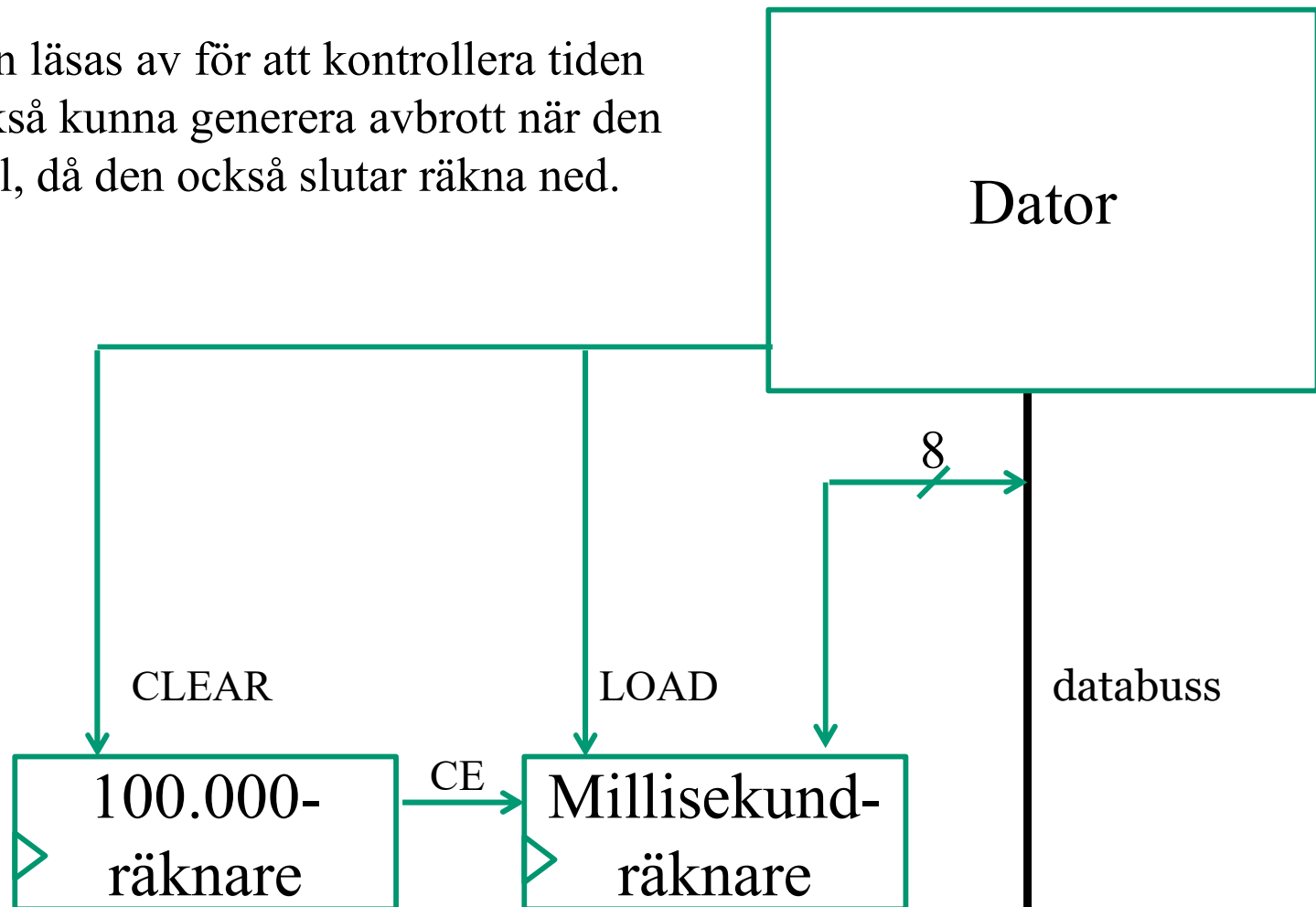


Timing

- Stora delar av tiden kommer erat program sannolikt bara att behöva vänta på att en viss tid har förflutit. Processorn klockas i 100 MHz och den kommer alltså att köra flera miljoner instruktioner per sekund. Så hur ska man göra för att få t ex ett grafisk objekt att förflytta sig med lämplig hastighet?
- Funkar det att göra timing i samband med bilduppdatering?
I så fall kanske Vsync kan användas på nåt sätt
- Kan man ha en vänteloop i programmet?
Hur många programcykler blir det?
- Kan man ha en separat räknare som håller reda på en viss tid?

Timing

- En timer-modul kan via I/O eller adressavkodning laddas med t ex ett millisekundvärde som sedan räknar ned under tiden som det övriga programmet fortsätter.
- Timerns värde kan läsas av för att kontrollera tiden
- Timern skulle också kunna generera avbrott när den räknat ned till noll, då den också slutar räkna ned.



Arbetsgång

- Undersök på bredden, 2-3 kanske 4 dagar
 - Hur funkar joysticken, tangentbordet m m ?
 - Hur implementerar man ett block-RAM?
 - m m ...
- Implementera och testa en **LITEN** del åt gången
 - Gör modul i underkatalog
 - Skriv testbänk och simulera
- Sätt samman moduler **steg för steg**
 - Kopplas samman moduler på en högre nivå
 - Skriv testbänk och simulera
- Var inte rädd för att **riva upp och göra** om när lösningen fungerar dåligt

Planera – Agera – Reflektera - Korrigera

Statusrapport

- Under det pågående projektet nästa period kommer varje grupp att utföra enklare statusrapportering varje vecka.
- Syftet är dels att se att en lämplig/rimlig mängd med arbete görs i projektet, dels för att handledare lättare ska kunna se hur arbetet fortskrider och dels för att öva på att hålla ordning på förbrukad tid. Statusrapportering kommer att användas i flera kurser framöver och är ett vanligt sätt inom projekt för att mäta kostnad och planera resurser.
- I Gitlab under katalogen "statusrapport" finns en markdown-fil som man varje vecka redigerar och svarar på tre frågor:
 - Vilka framsteg har gjorts sedan förra rapporten?
 - Finns det några problem?
 - Vad ska göras kommande vecka?
- En påminnelse kommer i början av nästa period.