Datorkonstruktion TSEA83
Fö 11

ALU

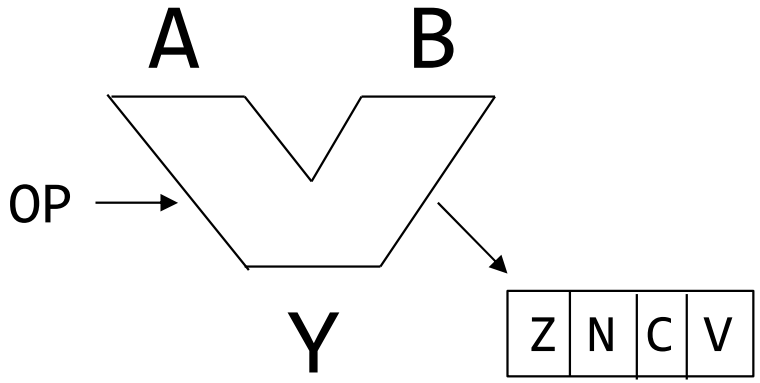
LINKÖPINGS UNIVERSITET

Datorteknik Fö : Agenda

- ALU
- Adressavkodning
- Avbrott
- Hierarki
- Arbetsgång
- Designskiss : Tips : Boot-loader, timing.
- Tidrapport

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:

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

$$N=y_3$$

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

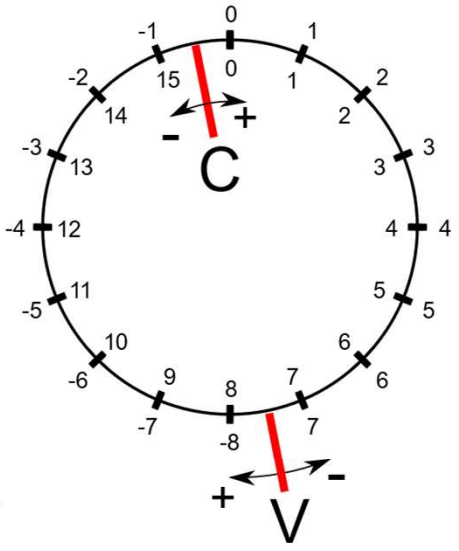
Dvs, ALU:n gör alltid på samma sätt.

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

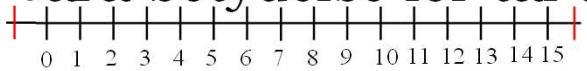


add medurs
sub moturs

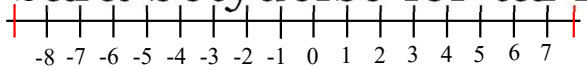
C 1-ställs när man passerar röda linjen från 15→0 (vid add), eller 0→15 (vid sub).

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

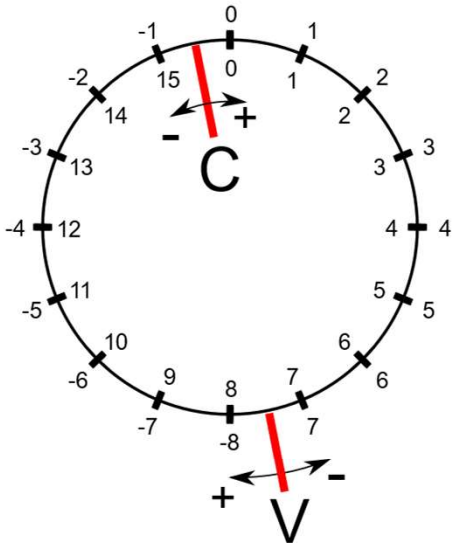
```
 0011    3    3
+1100   -4   12
01111   -1   15
Z:0,N:1,C:0,V:0
```

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

Subtraktion

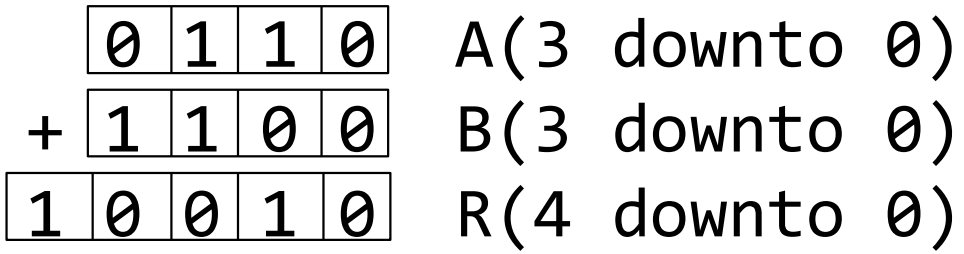
```
 0011    3    3
-0101    5    5
11110   -2   14
Z:0,N:1,C:1,V:0
```

```
 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




```
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)

```
        0 0 1 1   3
      * 1 0 0 1   9
      -----------
        0 0 1 1   1*0011
      0 0 0 0     0*0011
    0 0 0 0       0*0011
+ 0 0 1 1   (=24) 8*0011
-----------------
0 0 0 1 1 0 1 1   27
```

muls (med tecken)

```
          0 0 1 1   +3
        * 1 0 0 1   -7
        -----------
0 0 0 0 0 0 1 1     1*0011
0 0 0 0 0 0 0 0     0*0011
0 0 0 0 0 0 0       0*0011
+ 1 1 1 0 1   (=-24) -8*0011
-------------------
1 1 1 0 1 0 1 1     -21
```

Algoritmerna blir olika!

ALU : mul, muls, mulsu

Flaggorna Z, N, C, och V

mulsu (med/utan tecken)

```
          1 0 0 1   -7
        * 1 0 0 1    9
  1 1 1 1 1 0 0 1    1*1001
  0 0 0 0 0 0 0 0    0*1001
  0 0 0 0 0 0 0      0*1001
+ 1 1 0 0 1 0 0 0    8*1001
  (=-56)
  1 1 0 0 0 0 0 1   -63
```

mulus (utan/med tecken)

```
          1 0 0 1    9
        * 1 0 0 1   -7
    1 0 0 1 0 0 0 1   1*1001
    0 0 0 0 0 0 0 0   0*1001
  0 0 0 0 0 0 0 0     0*1001
+ 1 0 1 1 1 0 0 0     -8*1001
  (=-72)
  1 1 0 0 0 0 0 1   -63
```

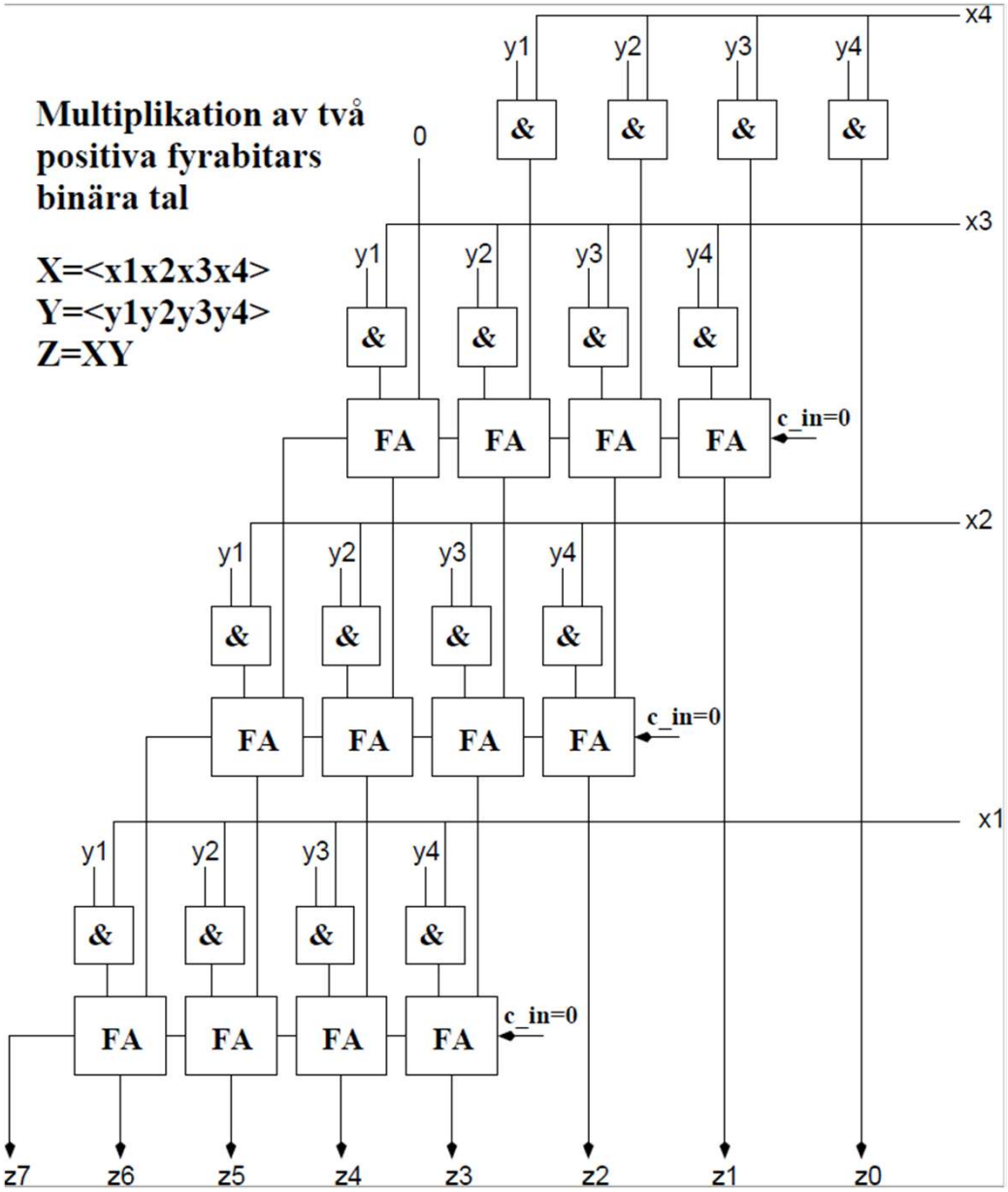
Ytterligare fler algoritmer!

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 s k exception error, dvs ett undantag som vanligen innebär att ett särskilt avbrott sker.

ALU : div, divs, (divsu)

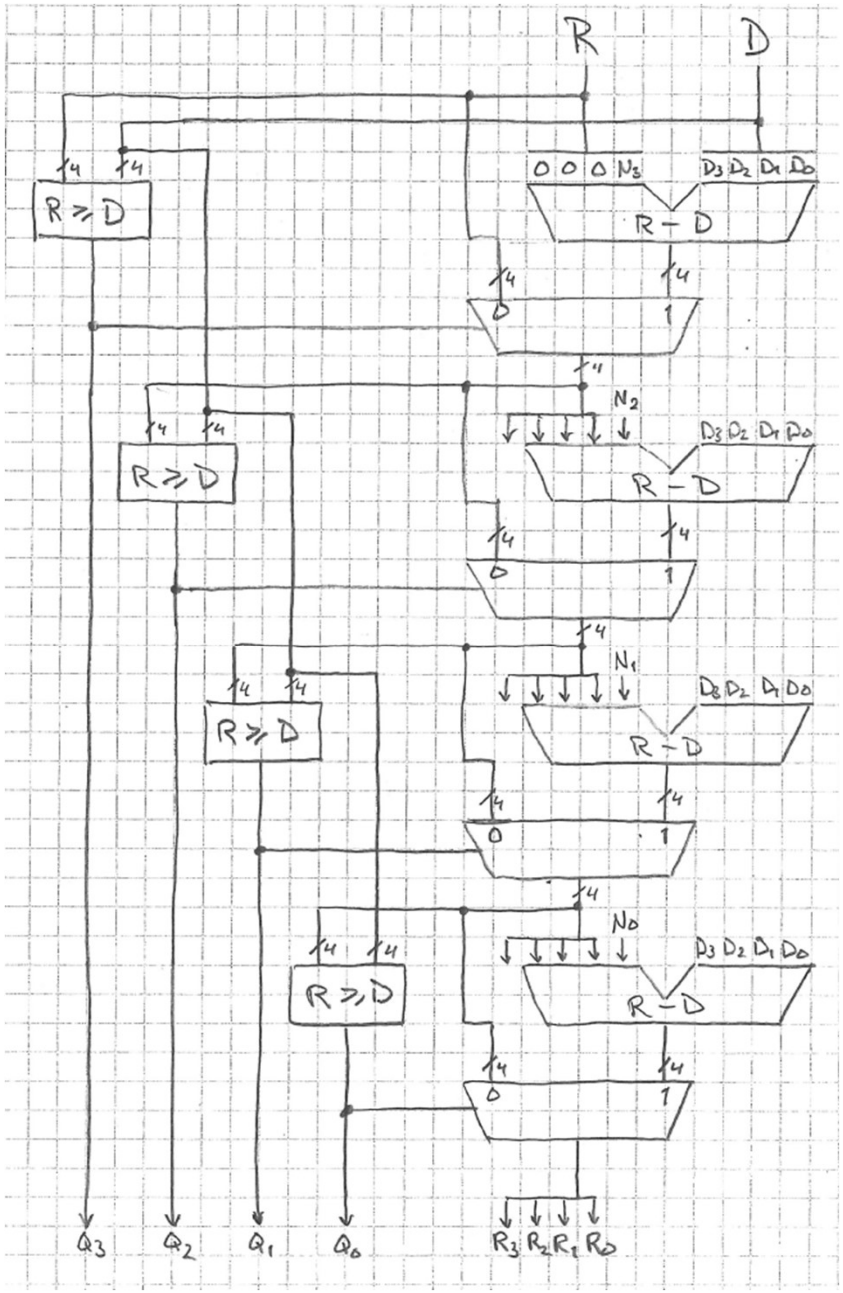
Flaggorna Z, N, C, och V

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
2. Beräkning av flaggor
3. Tilldelning av flaggor

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
...
```

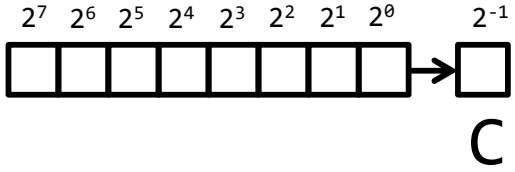
Man kan använda bara ADC men måste då alltid nollställa Carry innan ALLA additioner (även bara 4-bitars) för att resultatet säkert ska bli korrekt.

Pseudokod ”8-bitars addition” med bara ADC

```
CLC       ; Clear carry
ADC $D,$5 ; $D+$5     = $12
ADC $6,$2 ; $6+$2+$1 = $09
```

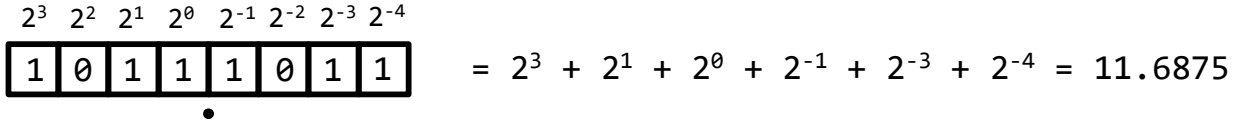
ALU : Fixtal

Hur representerar datorn tal med decimaler?
Vad händer när ett tal shiftas åt höger (delas med 2):



Minst signifikant bit åker ut i carry,
och är värd ½ = 2^{-1}

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:



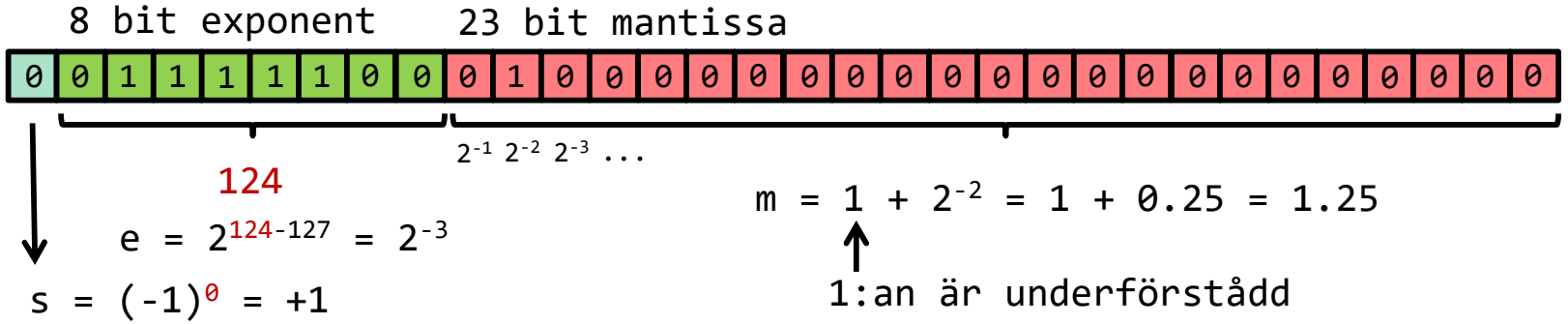
$= 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4} = 11.6875$

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 $0.15625 = 1.5625*10^{-1}$, och 1.5625 kallas mantissa och -1 kallas exponent. Talet kan lagras med basen 2 enligt $mantissa*2^{exponent}$, 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:



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

ALU : Flyttal

Hur adderar man två flyttal?

Antag: `3.5 + 0.15625`

$3.5 = \mathbf{1.}75 * 2^{1}$ (lagrat som flyttal)
$0.15625 = \mathbf{1.}25 * 2^{-3}$ (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.

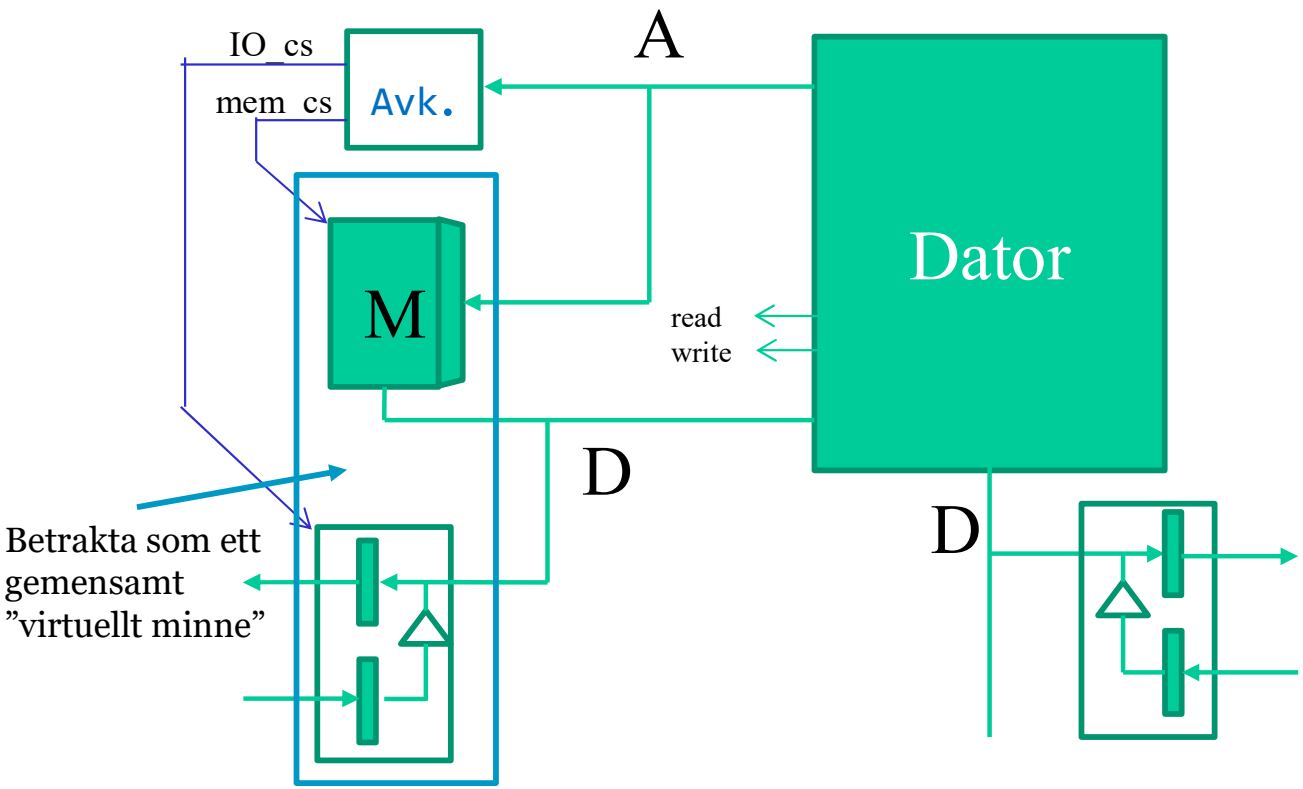
Det går förstås att bygga en special-ALU, alltså en BCD-ALU, men den får då ganska begränsad användning.

Datorkonstruktion TSEA83

Adressavkodning
Avbrott
Hierakisk konstruktion
Boot-loader
Timing
Arbetsgång
Tidrapport

LINKÖPINGS UNIVERSITET

Adressavkodning eller I/O-mappning?



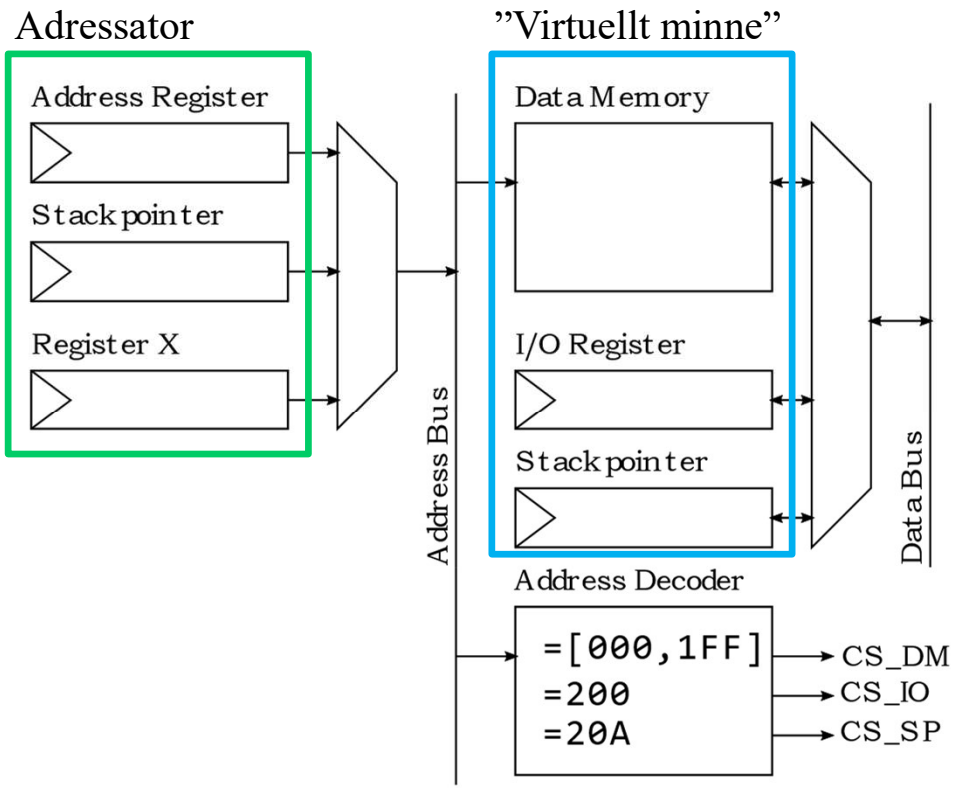
Memory-mapped I/O

- 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




```
ld r3,$20A   ; load SP to r3
st $30,r7    ; store r7 in DM
ld r5,$200   ; load IO to r5
st $1F0,r5   ; store r5 in DM
ld r3,$70    ; load r3 from DM
st $200,r3   ; store r3 in IO
```

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;
```

Avbrott

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 / töm pipelinen
- Hoppa till avbrottsvektor
- Kör avbrottsrutin, ISR.
- Återhopp från avbrott
- Återställ spärrvippan=0

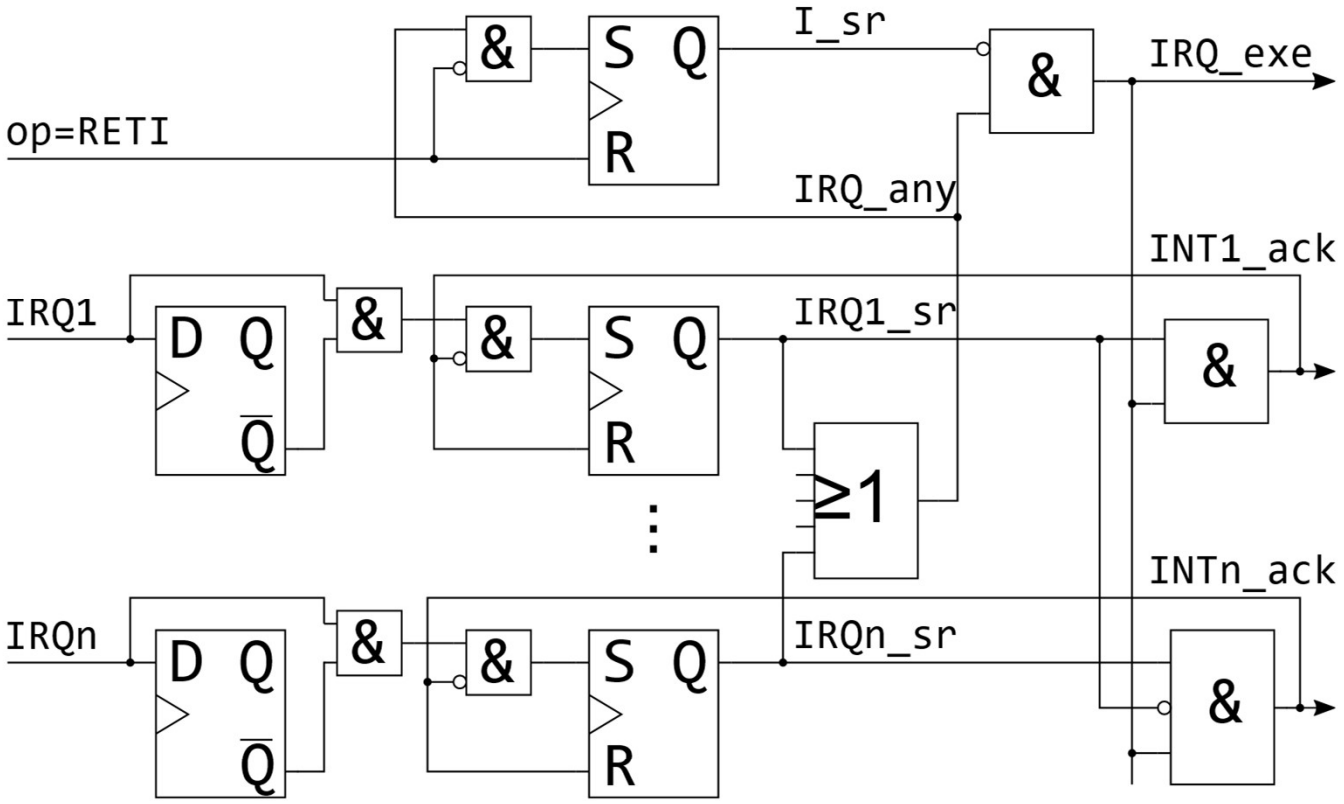
Flera avbrottskällor kräver mer funktionalitet.
-Ska avbrott prioriteras, dvs att avbrott kan avbryta varandra?
-Hur noterar man andra avbrott under ett pågående avbrott?

Avbrott

Tänkbar avbrottsmekanism för flera avbrottskällor

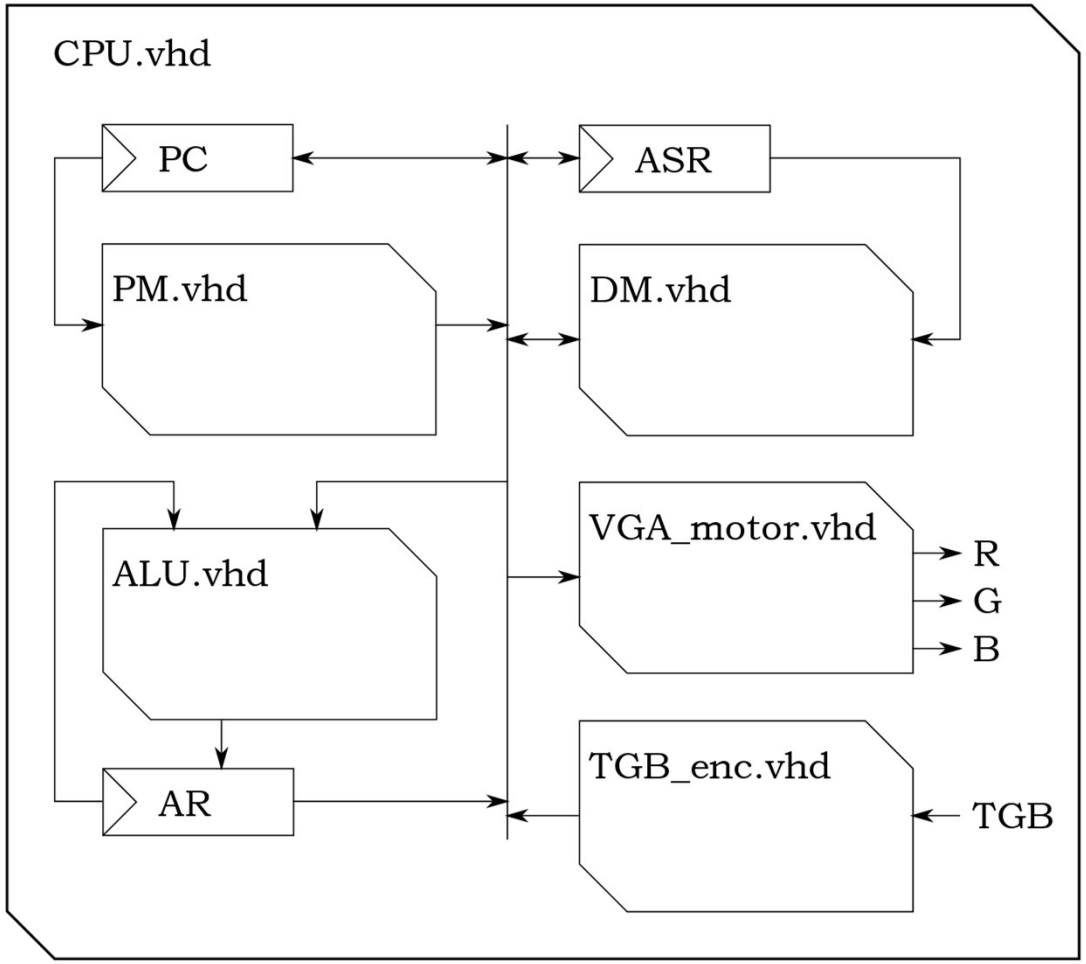
Egenskaper:

- Flera möjliga avbrotts-källor
- Pågående avbrott avbryts inte
- Andra avbrott registreras under pågående avbrott
- Bestämd prioritets-ordning vid samtidiga avbrott



I_sr : utsignal spärrvippa (0=avbrott möjligt, 1=avbrott pågår)
IRQ_exe : något avbrott påbörjas
IRQx : avbrottsbegäran x
IRQx_sr : avbrottsbegäran registrerad
IRQ_any : reg. avbrottsbegäran från något (av flera) avbrott
INTx_ack : avbrott x bekräftad (påbörjad)

Hierakisk / modulbaserad konstruktion och katalogstruktur



Katalogstruktur

```
/proj/CPU.vhd
/proj/CPU_tb.vhd
/proj/Makefile
/proj/Nexys3.ucf
/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
signal 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));

    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**!

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**!

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.

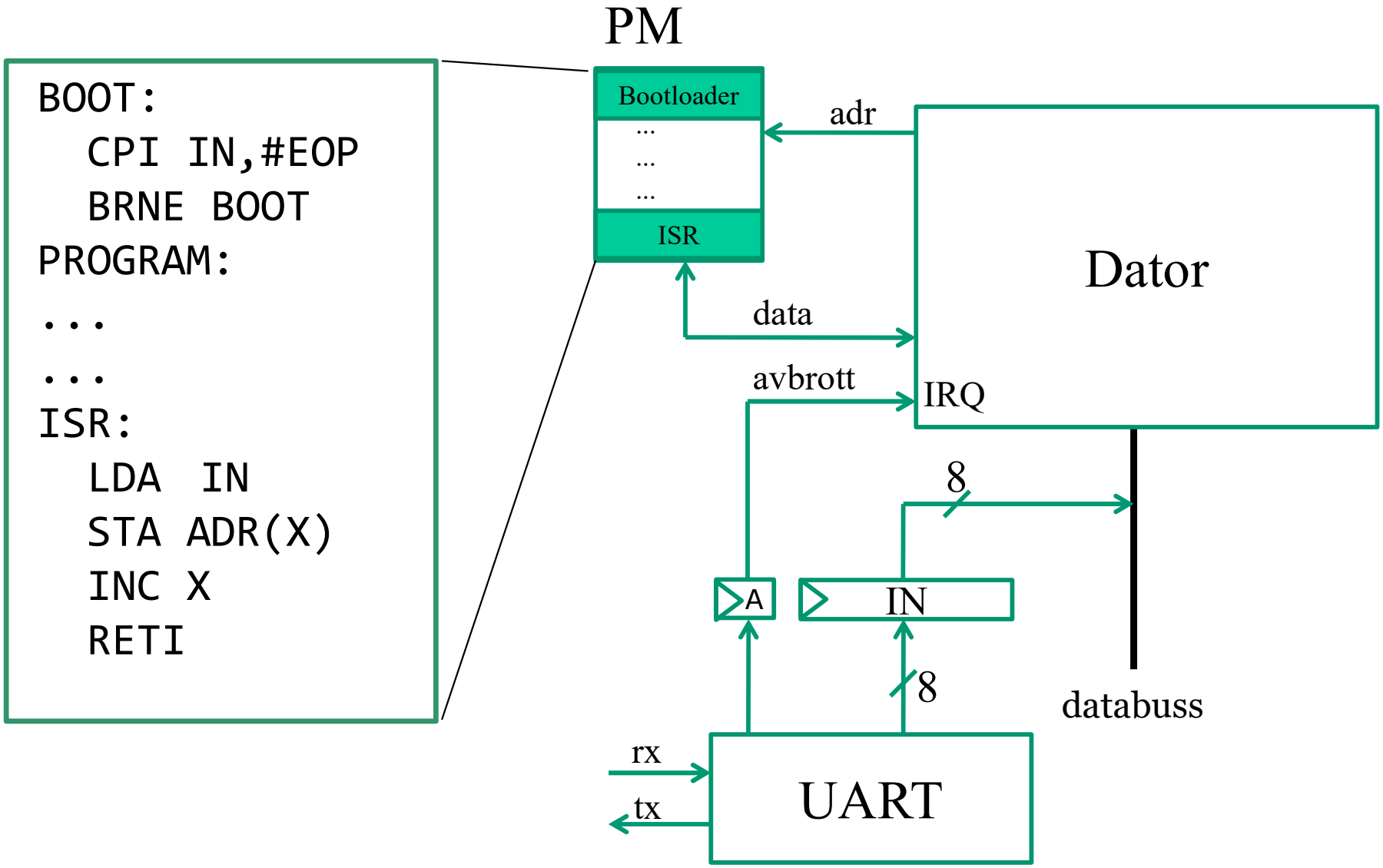
Det kan löna sig att bygga en boot-loader vars enda uppgift är att vänta på inkommande data/programkod från en UART, placera in det på rätt plats i minnet på den körande datorn och sedan starta programmet.

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

Boot-loader

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

```
BOOT:
   CPI IN,#EOP
   BRNE BOOT
PROGRAM:
...
...
ISR:
   LDA  IN
   STA ADR(X)
   INC X
   RETI
```



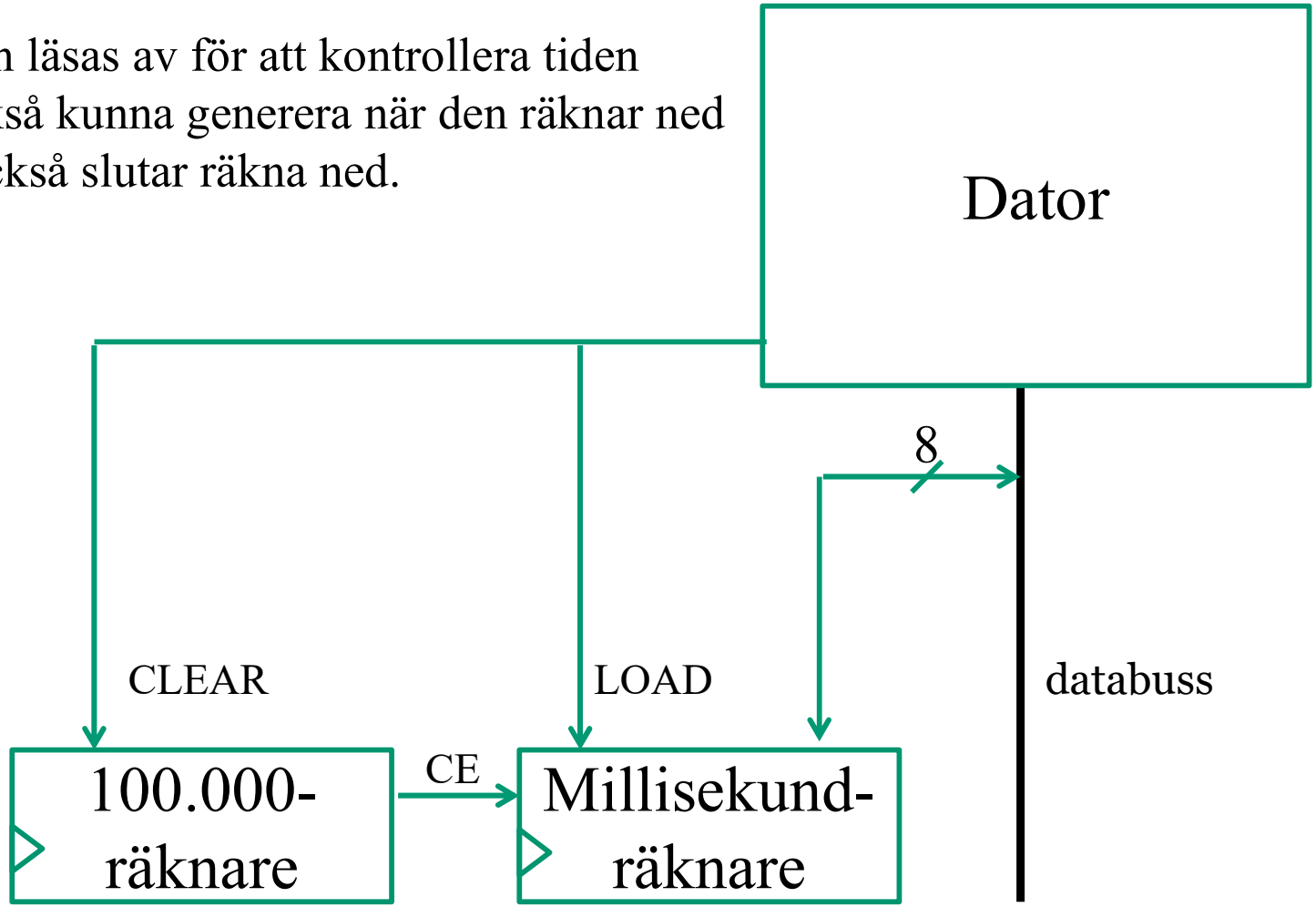
EOP = kod för End Of Program

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 när den räknar 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 – **A**gera – **R**eflektera - **K**orrigera

Tidrapport

- Under det pågående projektet nästa period kommer varje grupp att utföra enklare tidrapportering 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. Tidrapportering kommer att användas i flera kurser framöver och är ett vanligt sätt inom projekt för att mäta kostnad och planera resurser.
- Excel-arket för tidrapportering finns under Files i kanalen General i Teams. Ladda ner Excel-arket och spara tillsammans med övriga dokument i Gitlab.
- En påminnelse kommer i början av nästa period.