Datorkonstruktion TSEA83
Fö 1/4 kl 15:15-17:00

ALU

LINKÖPINGS UNIVERSITET

1

Datorteknik Fö : Agenda

- Informationskanaler
- Fö : ALU, Adressavkodning, Avbrott, Hierarki, Arbetsgång
- Git-konton för projektkod
- Arbeta hemifrån

2

Informationskanaler

- Zoom : Används till alla föreläsningar
  -Logga in via https://liu-se-zoom.us (ladda även ner appen här)
  -Meeting ID: 542 196 573 (Zoom-appen)
  -https://liu-se.zoom.us/j/542196573 (webbläsaren)
- Microsoft Teams : Används till projektmöten
  -TSEA83_VT2020
- Lisam : Används för labanmälan och inlämning av rapporter
- Mail : Används för övrig kommunikation
- Websidan : Används för kursmateriel och dylik information
  -http://www.isy.liu.se/edu/kurs/TSEA83/

3

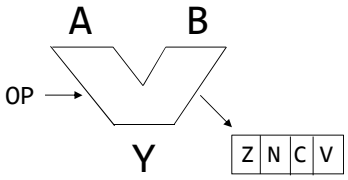
ALU

- ALU
- Adressavkodning
- Avbrott
- Hierarki / Katalogstruktur
- Arbetsgång

4

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

5

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.

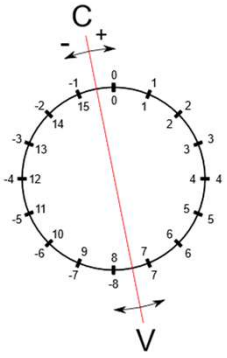
6

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

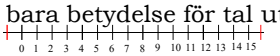


add medurs
sub moturs

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.

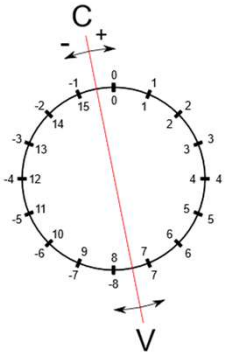
7

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
```

8

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';
```

9

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       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*0011
  0 0 0 0 0 0     0*0011
+ 1 1 1 0 1       -8*0011
  1 1 1 0 1 0 1 1 -21
```

Algoritmerna blir olika!

10

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*1001
  0 0 0 0 0 0        0*1001
+ 1 1 0 0 1          8*1001
  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    1*1001
        0 0 0 0      0*1001
      0 0 0 0        0*1001
+ 1 0 1 1 1          -8*1001
  1 1 0 0 0 0 0 1   -63
```

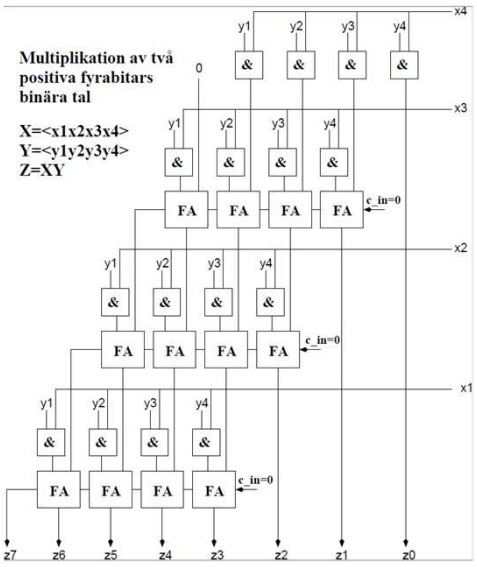
Ytterligare fler algoritmer!

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.

14

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.

15

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.

16

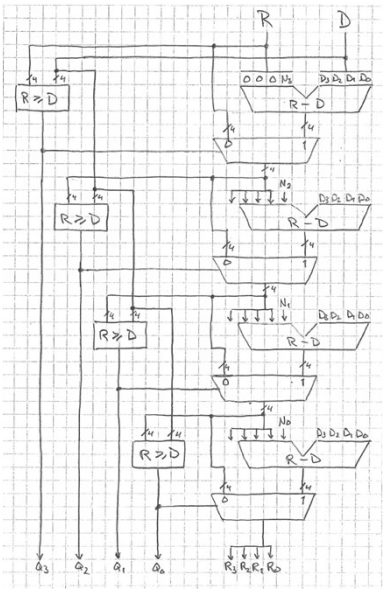
ALU : div, divs, (divsu)

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.



17

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.

18

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.

19

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);
H <= R(7 downto 4);
```

20

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';
```

21

ALU : VHDL-kod

3. Tilldelning av flaggor

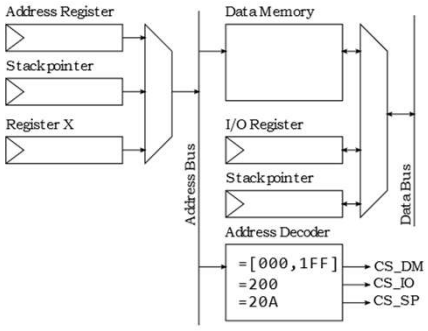
```
-- Z, N, C, V are the actual flags
signal Z, N, C, V : std_logic;

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.

22

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.

23

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;
```

24

Avbrott

Princip

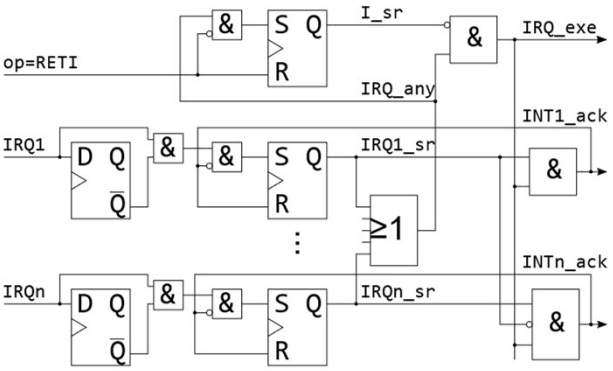
- Kontrollera om avbrott kan utföras (spärrvippan = 0)
- Sätt spärrvippan=1
- Kör färdigt pågående instruktion / töm pipelinen
- Hoppa till avbrottsvektor
- Under pågående avbrott, notera andra avbrott
- Återhopp från avbrott
- Återställ spärrvippan=0

25

Avbrott

Tänkbar avbrottsmekanism

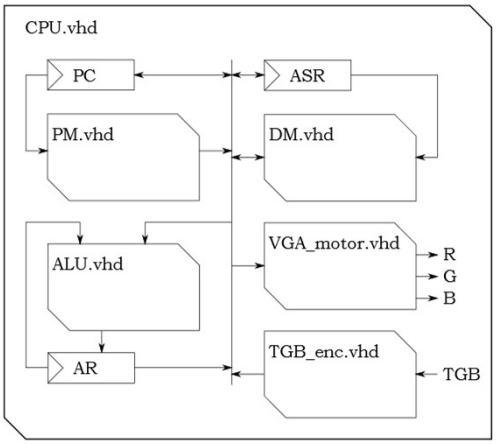
Egenskaper:
- Flera möjliga avbrotts-källor
- Pågående avbrott avbryts inte
- Andra avbrott registreras under pågående avbrott
- Bestämd prioritets-ordning vid samtidiga avbrott



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)

26

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

27

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.

28

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

29