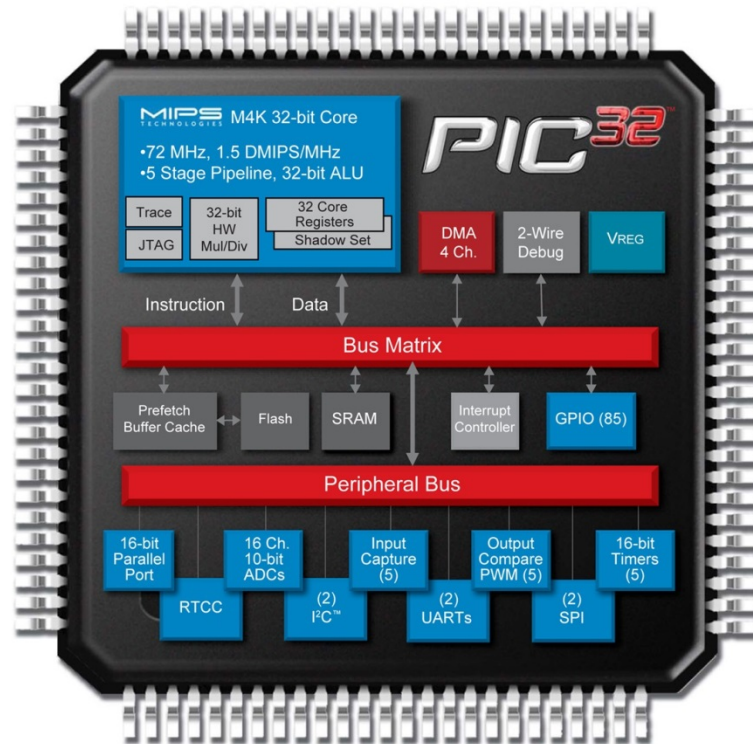


Varför använda en liten 8-bitars processor när det finns billiga kraftfulla 32-bitars?



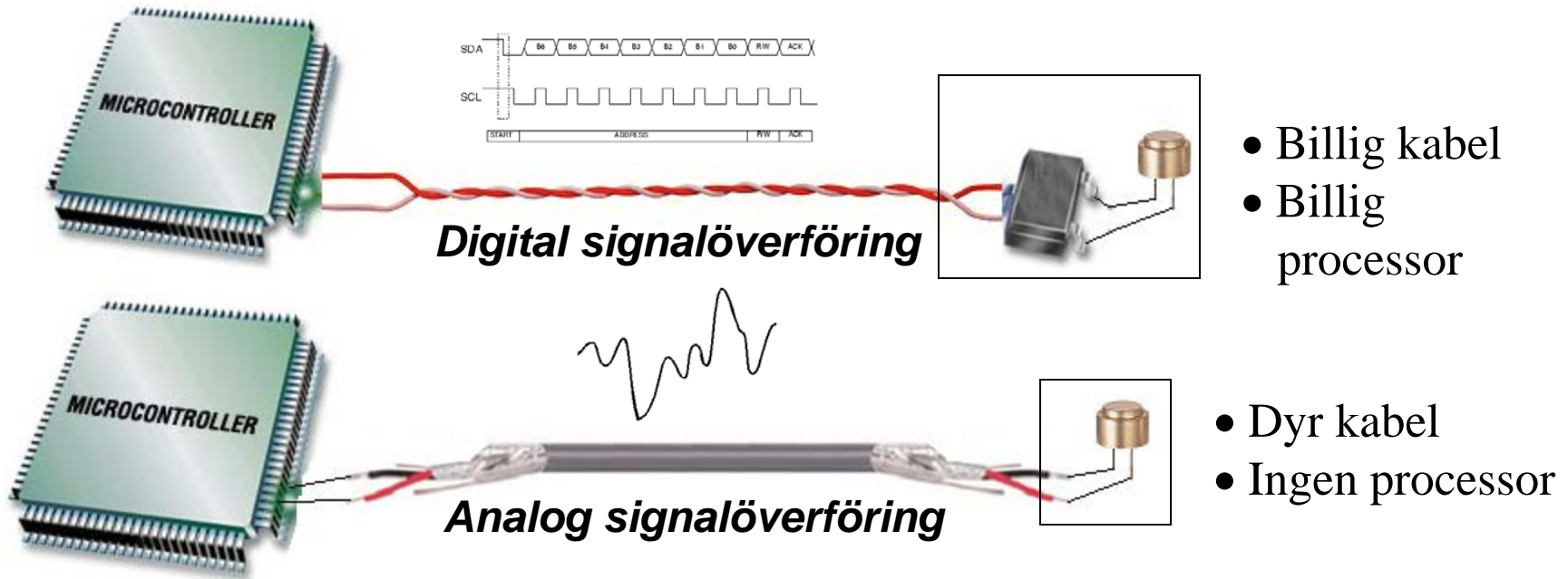
8-bits-processor nära givaren?

- En enkel givare har ofta en *svag utsignal*. Den kan behöva anslutas med en **dyr kabel**.
- En dyr givare med ”*inbyggd elektronik*” kan klara sig med en **enklare kabel**.

Kostnaden för båda alternativen kan i slutändan mycket väl bli densamma!

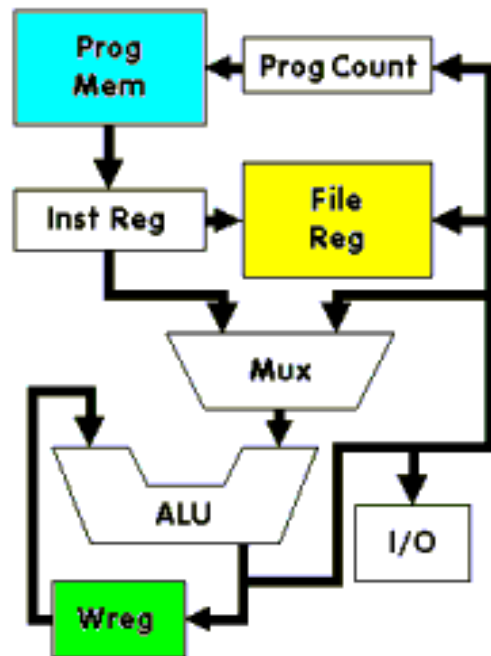
Således smart att bygga in en 8 bitars processor inuti givaren!

8-bits-processorn som kabel?



Hur *många* 8:a bitars processorer får man för *en meter* kabel? Processorn som kabelersättare!

PIC 8-bitsprocessor



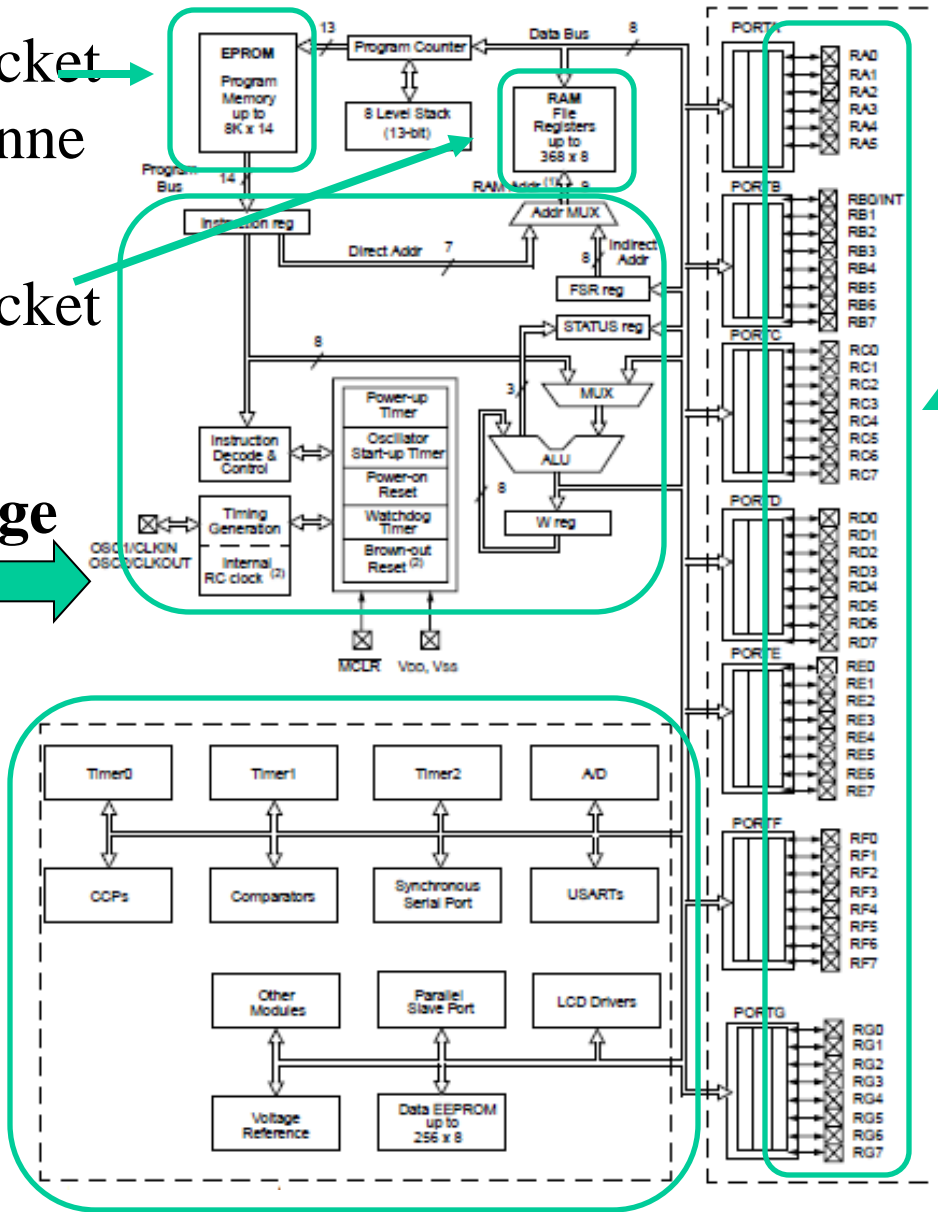
PIC (Peripheral Interface Computer) är billiga datorkretsar med "allt i ett".

- Olika mycket programminne

- Olika mycket data minne

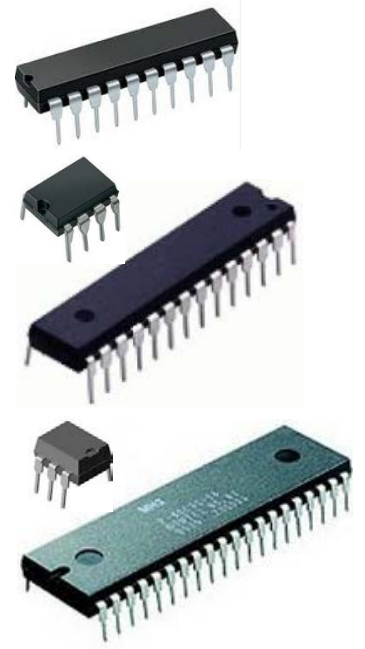
PIC Midrange processor

- Olika kombinationer av IO-enheter



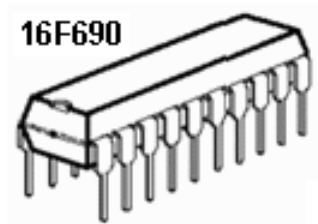
63 olika typer av Midrange PIC processorer!

- Olika många pinnar



Affärsidén – köp bara precis så mycket Du behöver

Utveckla din tillämpning på en processor som har ”litet av allt”.



Till den färdiga produkten använder man sedan bara precis så mycket man behöver.



ELFA's *billigaste* PIC-processor



73-874-42

Microcontroller 8 Bit SOT-23-6

1

Köp

1- 4.75

10- 4.00

50- 3.63

Microchip

PIC10F220T-I/OT

4 kr st om Du köper 10 st ...

Programminne: 384 ord
RAM-minne: 16 Byte
8 bit AD-omvandlare 2 kanaler
Inbyggd oscillator 4 MHz
TIMER0
Spänning 2...5,5 V
Typisk strömförbrukning: 175µA

PIC10F220T-I/OT

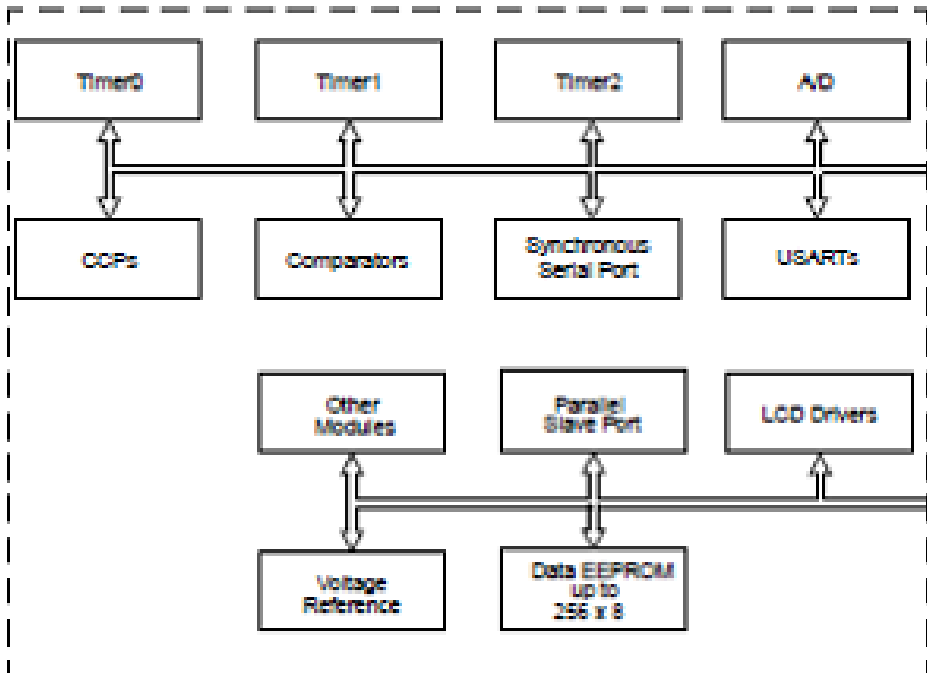
Kan kompileras med

Cc5x – includefil finns

När datorkraft är så här billig öppnas helt nya möjligheter ...

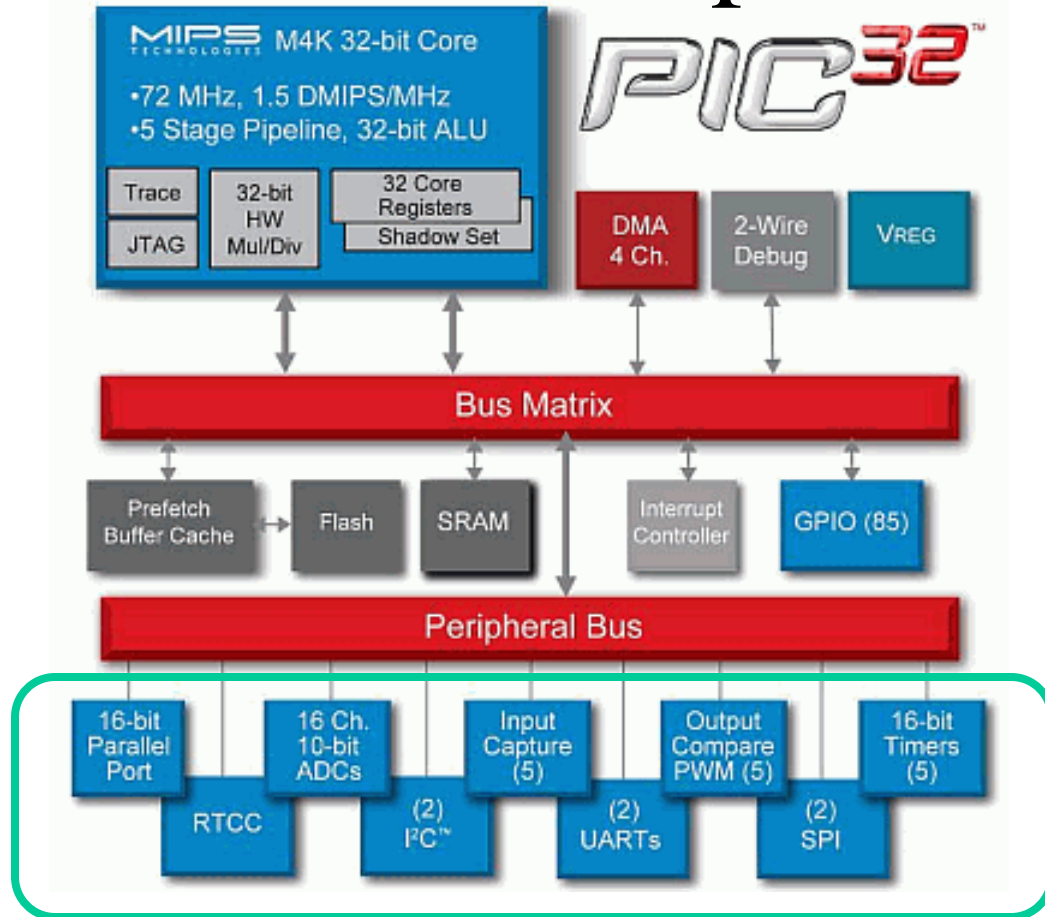
Detta är ett skäl till att det kan vara bra att lära sig PIC-processorer!

De inbyggda IO-enheterna ökar 8-bitsprocessorernas prestanda



IO-portar och IO-bitar,
Timers,
Capture/Compare/PWM,
Analoga komparatorer,
AD-omvandlare,
Serieportar,
Spänningsreferenser, Data
EEPROM mm

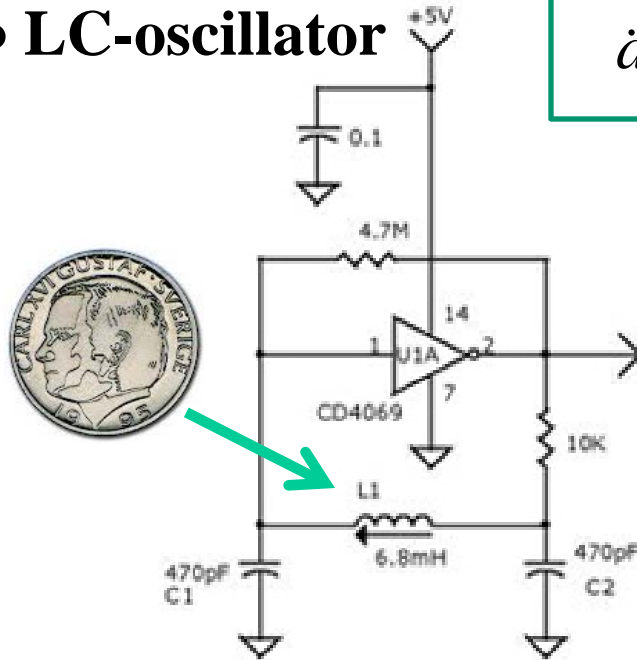
Samma IO-enheter återfinns sedan även i större processorer



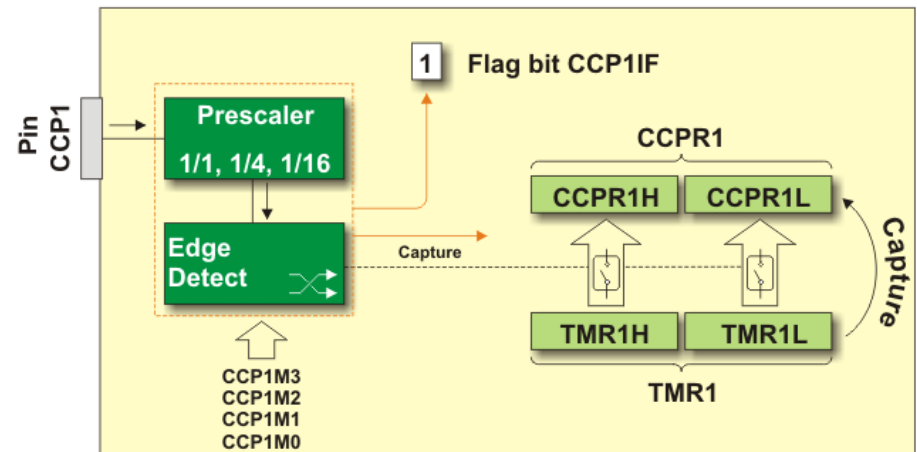
Kursen handlar om att ansluta elektronik till **IO-enheter**

Tex. Hur indikerar man att enkronan är i närheten (av spolen)?

- **LC-oscillator**



- **CCP-enhet**



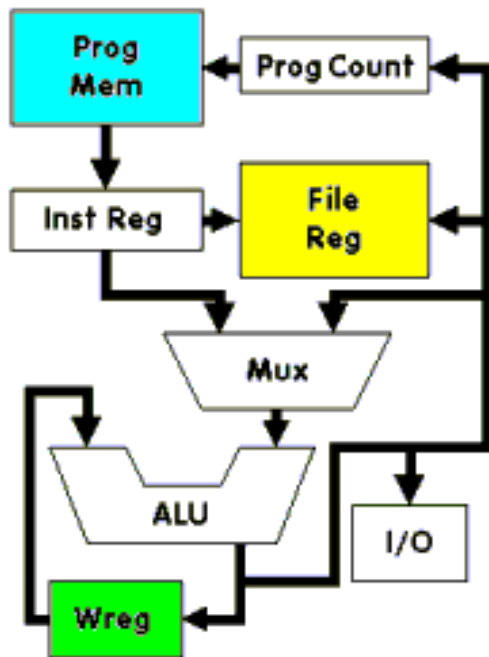
Kretsteori och PIC-processor!

Du kommer tillexempel att få veta hur en induktiv givare fungerar ...



PIC16F690

PIC 8-bitsprocessor

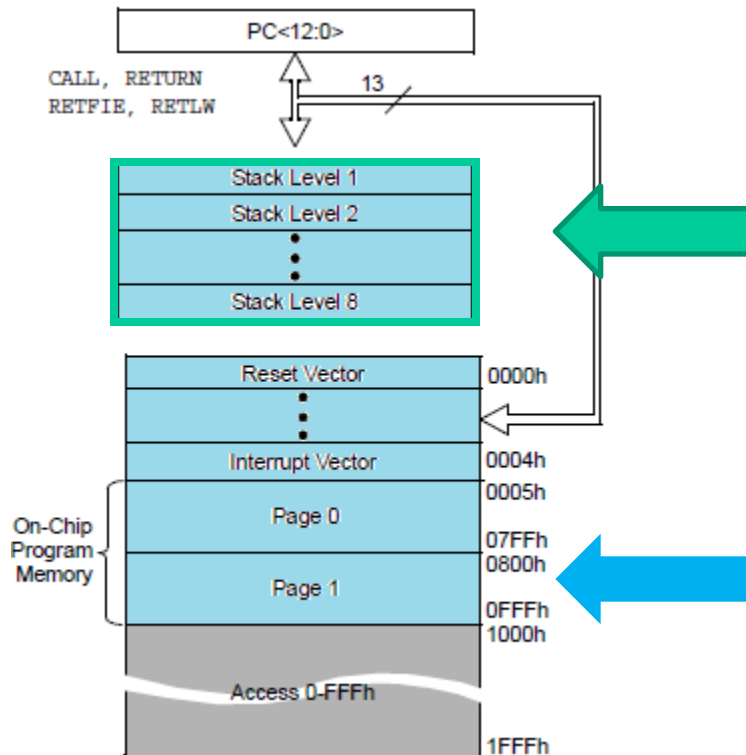


PIC (Peripheral Interface Computer) är billiga datorskretsar med "allt i ett".

Prog Mem. Programminnet.

File Reg. Dataminne och specialregister. Specialregistren är kopplade till IO, tex. pinnarna.

Program memory



Stack
bara för återhopp-
adresser (8st), inte
parametrar.

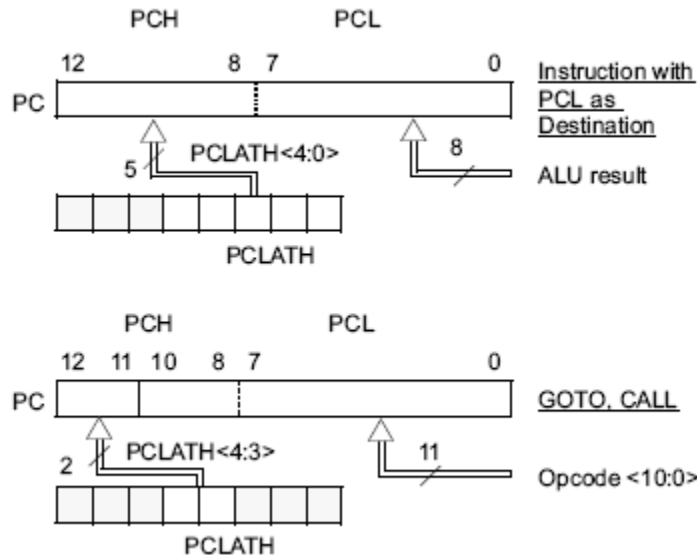
Programminne.
PIC16F690 har 7 kByte
FLASH.

4096 word a' 14 bitar.

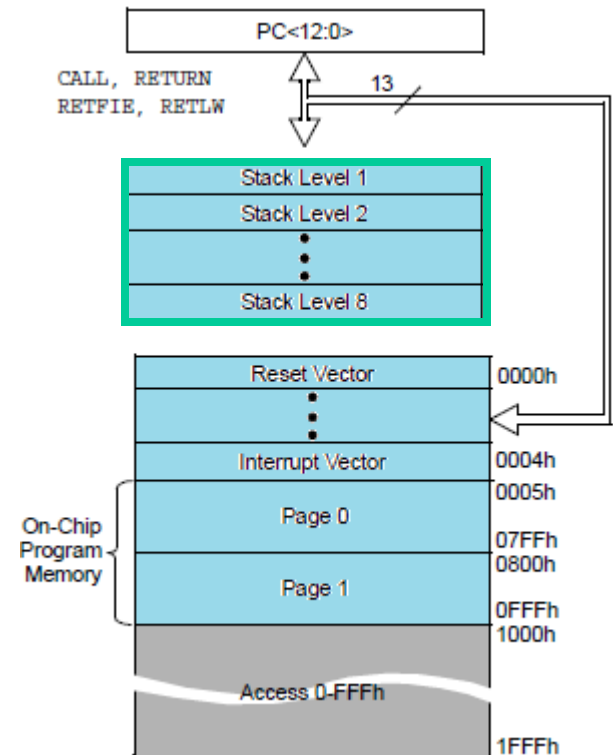
16F690 Programminne

PIC-processorernas **GOTO** och **CALL** - instruktioner når direkt alla adresser inom **2 k** (opkoden har **11** adressbitar).

16F690 har **4 k** programminne, så därför kan man behöva välja ny ”sida” i program-minnet.



Uppdelningen i sidor, pages, är en omodern arkitektur.



Codepages

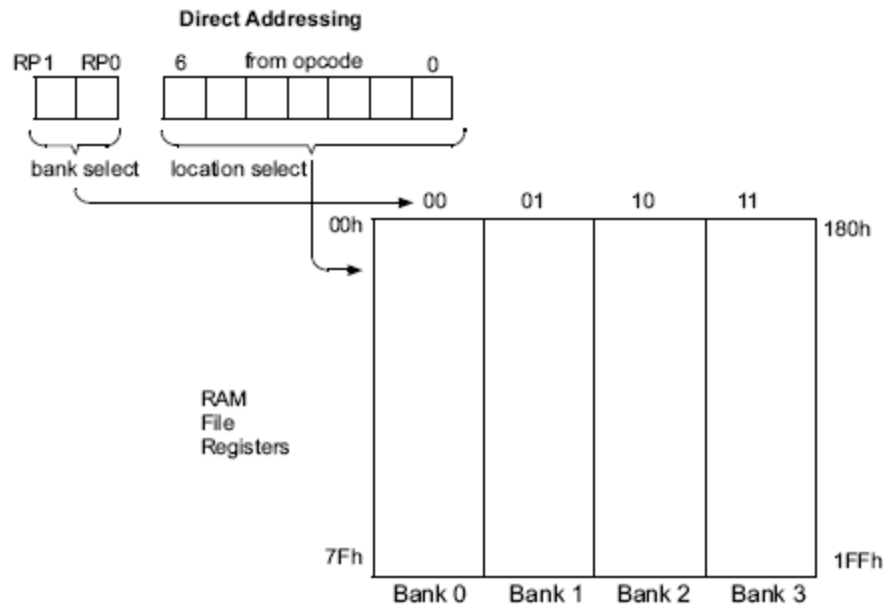
PIC-processorerna har programminnet uppdelat i "sidor" (0, 1, 2, 3), codepages, om 2048 instruktioner.

Kompilatorn **Cc5x** börjar att lägga ut kod på **page 0** och ger felmeddelande om denna sida inte räcker till. Skulle detta inträffa så skriver man dit instruktionen **#pragma codepage 1**, så att alla instruktioner därefter hamnar på nästa sida (och så vidare **codepage 2** vid behov).

För att få kompakt kod behövs en noggrann "sidplanering", något som man knappast bryr sig om vid prototyputveckling.

RP1 och RP0

Bank väljs med bitarna **RP1** och **RP0** i **STATUS** registret



	RP1	RP0
Bank0	0	0
Bank1	0	1
Bank2	1	0
Bank3	1	1

Uppdelningen i RAM-bankar är en omodern arkitektur.

Kompilatorn kan välja åt oss!

PIC-processorernas registerarea (RAM-minne) är uppdelat i "rambankar" (0, 1, 2, 3). **Cc5x** börjar att fylla rambank 0.

Man kan byta rambank med instruktionen

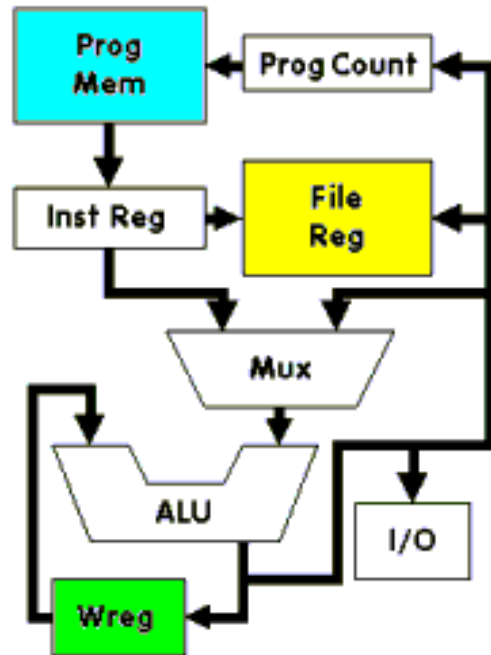
#pragma rambank 1 och då placeras alla variabler som deklarerats därefter ut i nästa rambank (rambank 1).

En del minnesceller återfinns på samma plats i alla rambankar, så kallad *mapped RAM*. Man kan välja att placera variabler i "mappad ram" (så länge det finns plats) med instruktionen

#pragma rambank - .

Bästa användningen av RAM-bankar kräver mycket planering, något som man knappast behöver bry sig om vid prototyputveckling.

PC, IR, ALU, W-registret



Prog Counter, PC. Programräknarregistret pekar ut var i programmet man är. Det räknas upp automatiskt efter varje utförd instruktion.

Inst Register, IR.

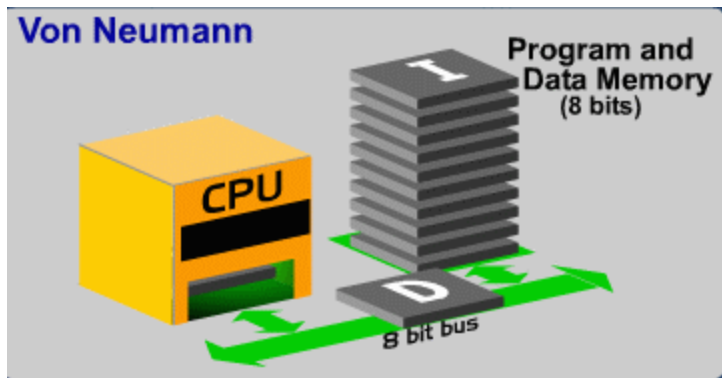
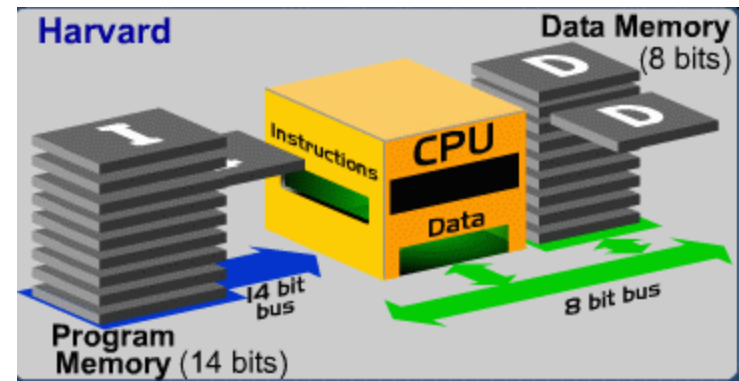
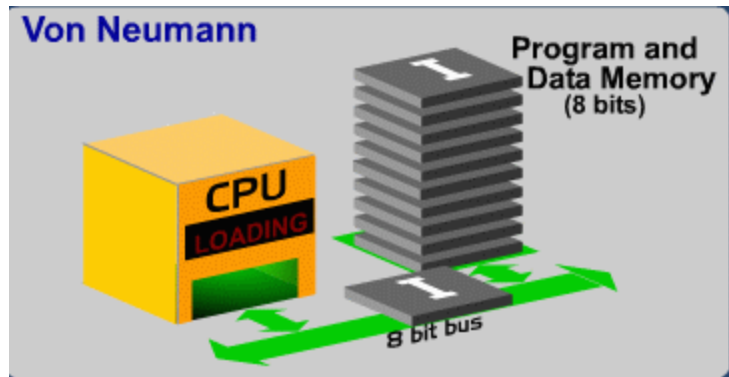
Instruktionsregistret håller koden för aktuell instruktion.

ALU. Aritmetisk Logisk Enhet sköter beräkningarna.

De allra flesta operationerna går igenom arbetsregistret, **W-reg**. Detta är PIC-processorns ”getingmidja”.



Harvard vs Von Neumann



- **Von Neumann** arkitekturen har en *gemensam* buss för instruktioner och data.
- **Harvard** arkitekturen har *skilda* bussar för instruktioner och data.

Harvard är (dubbelt) snabbare ...

CISC vs RISC

- **CISC** (Complex Instruction Set Computer)
Ex. Intel PC, som har **700** instruktioner.
- **RISC** (Reduced Instruction Set Computer)
Ex. Microchip PIC, som har **33** instruktioner.

Dessa begrepp är idag föråldrade. Intel-processorerna klassificeras visserligen fortfarande som CISC – men de har avancerad arkitektur som utnyttjar det bästa från RISC ...

KIA's fabrik i Slovenien

En bil i minuten lämnar bandet – tar det en minut att bygga en bil?

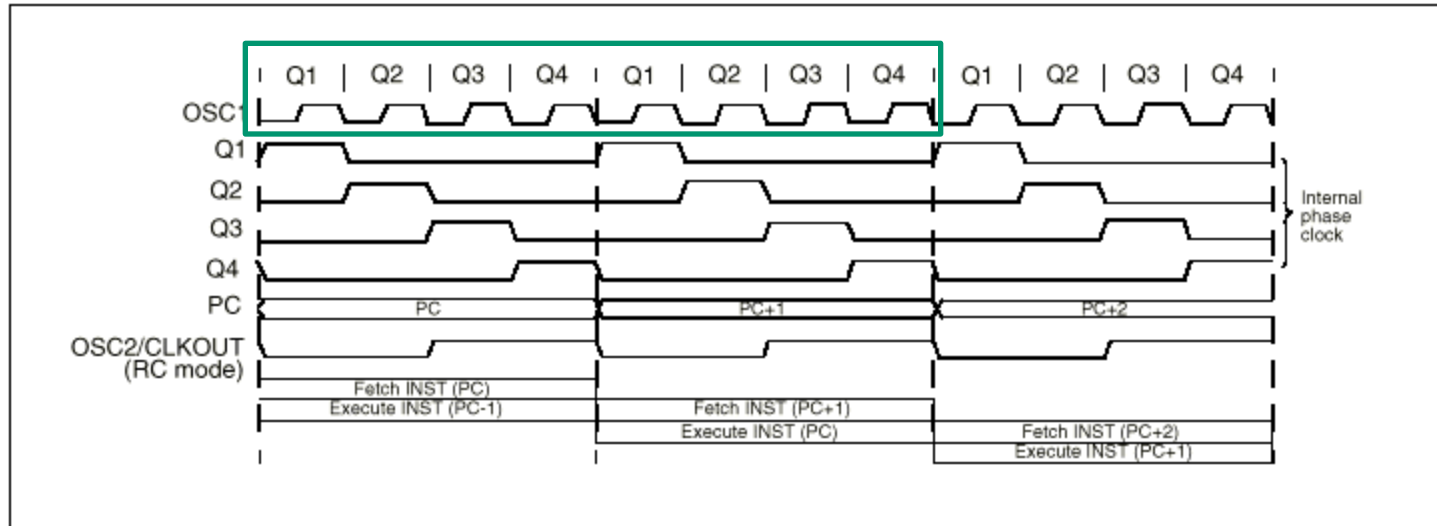
Nej för KIA's fabrik utanför Zilina tar det 18 mantimmar att bygga en bil (detta är ändå världsrekord! Toyota behöver c:a 30 mantimmar).



Lösningen är en **Pipeline**. 18 timmar är 1080 minuter, så bygget kan ske parallellt vid 1080 enminutersstationer. Fabriken har 3000 anställda som arbetar i treskift, dvs 1000 arbetare per skift. Många av stationerna är således helt robotiserade.

Fetch and Execute

FIGURE 3-3: CLOCK/INSTRUCTION CYCLE



PIC har Harvard arkitektur och kan därför hämta **Fetch** en instruktion *samtidigt* som den exekverar **Execute** en annan. Det tar **8 klockcykler** att utföra en instruktion. Vi har en **två stegs pipeline**, så det blir därför en instruktion *färdig* efter var fjärde oscillator-klockcykel.

Med 4 MHz klocka innebär det 1.000.000 instruktioner/sek. Varje instruktion tar således **1 μ s**.

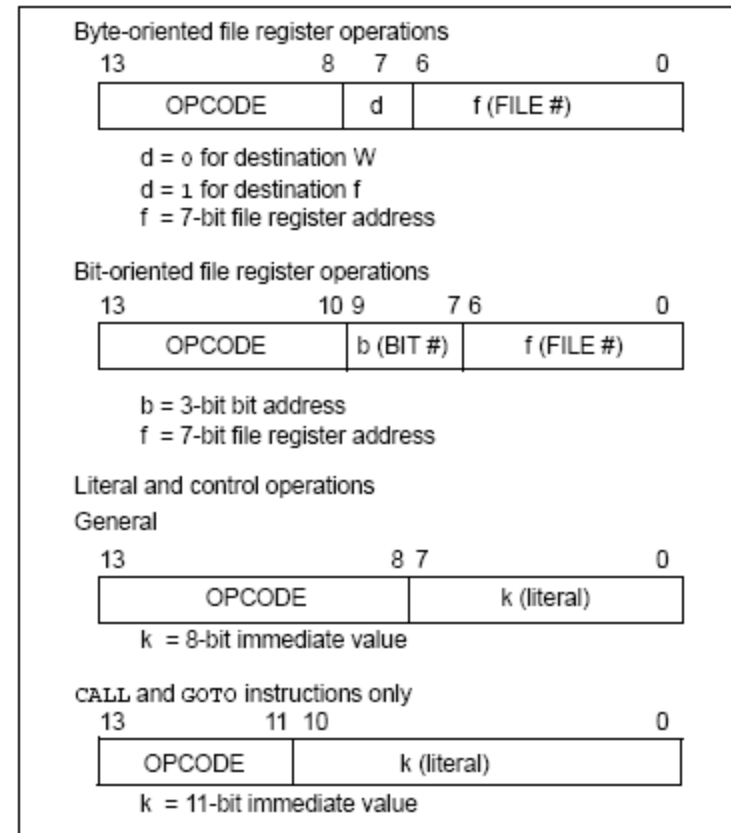
Instruktionsformat

PIC är en klassisk RISC-processor med bara 33 instruktioner ...

Instruktionerna är **14** bitar

- OP-koden *vad som ska göras* – är **6** bitar (eller 3 bitar).
- Resten av bitarna används för att tala om – *med vad* det ska göras.

FIGURE 15-1: GENERAL FORMAT FOR INSTRUCTIONS



Byte operationer

Ex. Addition av tal i **FILE**, dataminnet, och arbetsregistret **W**. Resultatet lagras i arbetsregistret eller dataminnet – och det ursprungliga talet skrivs över.

ADDWF f, d

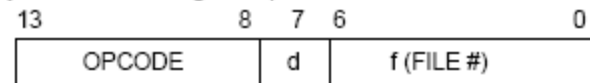
ADDWF $f, 0$; $W = f + W$

eller

ADDWF $f, 1$; $f = f + W$

På samma sätt: SUBWF f, d

Byte-oriented file register operations



d = 0 for destination W

d = 1 for destination f

f = 7-bit file register address

Assemblerinstruktioner skrivs som förkortningar **mnemonics**.

Fler Byteoperationer

Vissa specialfall av addition och subtraktion, *öka med ett* respektive *minska med ett*, har egna instruktioner. Liksom *0-ställning* av register.

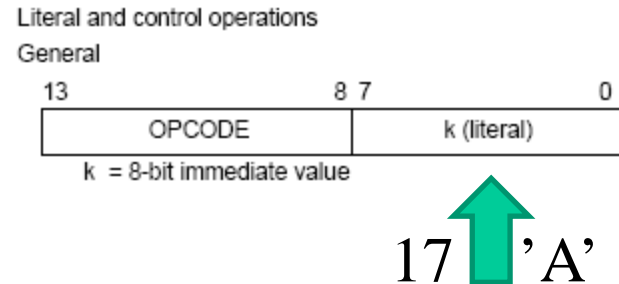
INCF **f,d** **DECF** **f,d** **CLRW** resp **CLRF** **f**

Vill man *kopiera* innehåll mellan minne och arbetsregistret gör man det med **MOVF** **f,0**; **W=f** eller mellan arbetsregistret och minnet med **MOVWF** **f**; **f=W**

Move betyder egentligen Copy!

Programkonstanter

Programkonstanter som talet 17 eller tecknet 'A' och liknande, lagras inuti instruktioner.

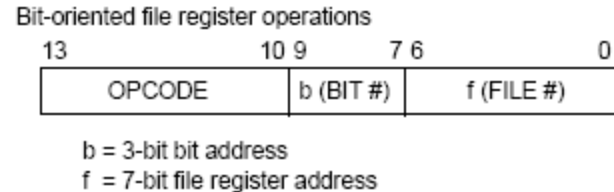


k är en ”**Literal**”, en Bytekonstant, som lagras inuti instruktionen **MOVLW k; W=k**. Vid exekveringen placeras konstanten i arbetsregistret.

Fler Literal-instruktioner: **ADDLW k; W=W+k**
SUBLW k; W=W-k

Bitoperationer

PIC-processorn har *direkta* bitoperationer.



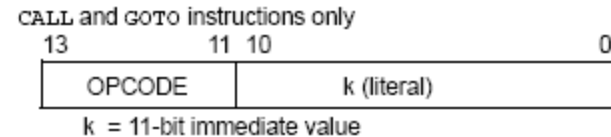
BCF **f**, **b** 0-ställ bit **b** i File nr **f** (bitarna numreras 0...7)
BSF **f**, **b** 1-ställ bit **b** i **f**

Programhopp

GOTO k Programhopp

CALL k Subrutinanrop

RETURN Återhopp



Vid **GOTO** ändras PC till värdet av Literal k som för denna instruktion är **11** bitar (och två extra bitar från register **PCLATH**). PC fortsätter exekvera programmet på *nytt* ställe.

Vid **CALL** sparas först PC:s värde i ett *stackregister*, därefter görs som vid **GOTO**. Vid **RETURN** hämtas PC:s värde tillbaka från stackregistret och programmet fortsätter med instruktionen som följer efter anropsinstruktionen.

Vilkortest, skip

PIC-processorn har några instruktioner som testar om vilkor är uppfyllda och i så fall *hoppas över*, skip, nästa instruktion. Nästa instruktion är då oftast en **GOTO** instruktion.

DECFSZ *f, d*; $f - 1$ men skip "nästa" om 0-resultat

INCFSZ *f, d*; $f + 1$ skip om 0 (register kan "slå runt"!)

BTFSC *f, b*; skip om bit *b* i *f* är 0 (Clear)

BTFSS *f, b*; skip om *b* i *f* är 1 (Set)

Detta bakvända tänkande "*låt bli att hoppa om...*" är lite speciellt för PIC och *inte* längre vanligt för andra processortyper.

Varför skip?

Utfallet av ett test innebär ofta att man behöver göra en *extra instruktion* som man annars inte skulle göra.

skip-instruktionen hoppar över denna extra instruktion, och eftersom hopp alltid tar dubbelt så lång tid som andra instruktioner så tar instruktionsföljden lika lång tid att exekvera oavsett testets utfall!

Detta kan ses som en finess med PIC-processorerens instruktionsuppsättning.

NOP No Operation

NOP	No Operation				
Syntax:	[<i>label</i>] NOP				
Operands:	None				
Operation:	No operation				
Status Affected:	None				
Encoding:	<table border="1"><tr><td>00</td><td>0000</td><td>0xx0</td><td>0000</td></tr></table>	00	0000	0xx0	0000
00	0000	0xx0	0000		
Description:	No operation.				
Words:	1				
Cycles:	1				
<u>Example</u>	NOP				

Processorer har i allmänhet en instruktion som gör ”ingenting”. Den kan läggas till för att utjämna tidskillnader mellan olika vägar i program.

Hur lång tid tar instruktionerna?

Processorns interna klocka arbetar med $\frac{1}{4}$ av kristallfrekvensen. Vanligt är därför 4 MHz kristall och då blir det 1 MHz klockfrekvens. De flesta operationer utförs på *en* klock-cykel dvs. tar **1 μ s**. De instruktioner som påverkar PC tar två klock-cykler dvs. **2 μ s**.

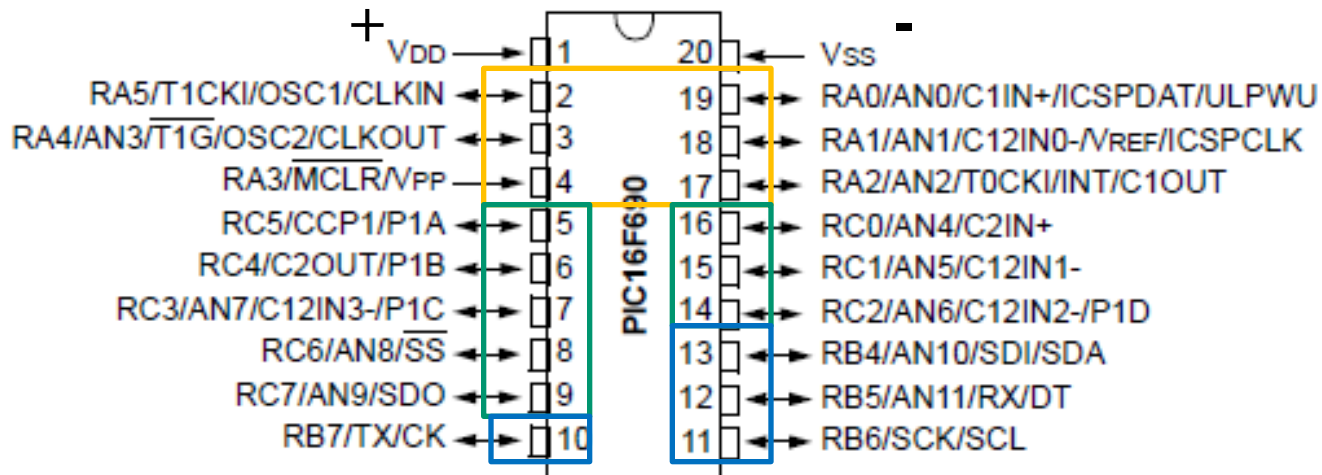
GOTO, **CALL**, **RETURN** tar alltid 2 cykler,

DECFSZ, **INCFZ**, **BTFSC**, **BTFSS** tar 2 cykler de gånger det blir "skip", annars 1 cykel.

Man kan räkna ut PIC-processorns exekveringstid med fingerräkning!



Portar



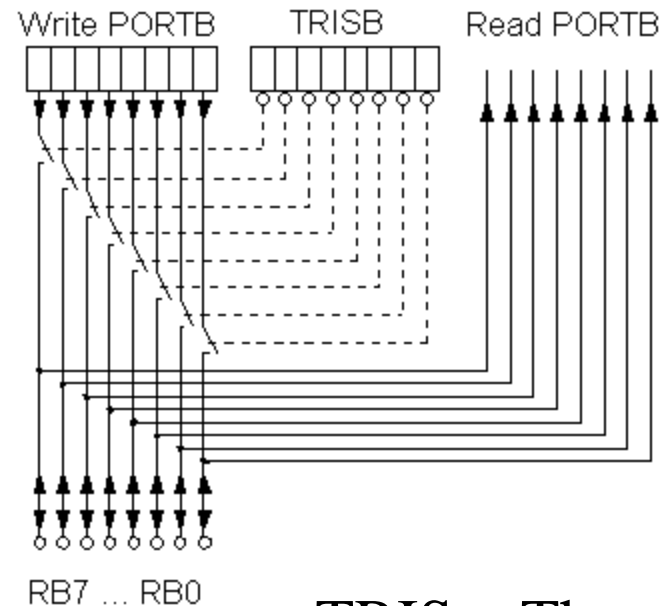
Av PIC-kretsens pinnar är 6 samlade till en **PORTA** och 8 till en **PORTC**, 4 till en **PORTB**. Pinnarna kan även användas ensamma, och som synes kan de ha många alternativa funktioner.

Tris-register

Om en pinne ska användas som **ingång** eller **utgång** avgörs av ett TRIS-register.

TRISA och **TRISB** och **TRISC**

Om ”*motsvarande*” bit i trisregistret är **1** används portpinnen som ingång, om den är **0** som utgång!



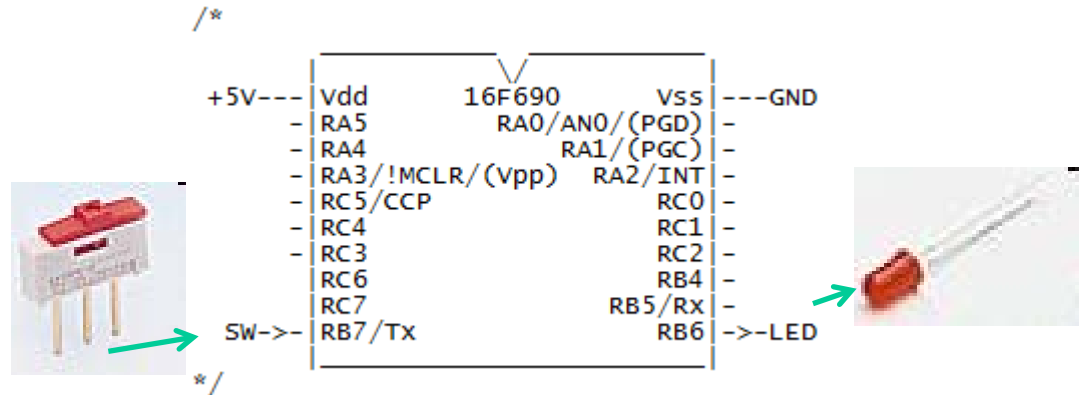
TRIS = Threestate

TABLE 15-2: PIC16F627A/628A/648A INSTRUCTION SET

Mnemonic, Operands	Description	Cycles	14-Bit Opcode				Status Affected	Notes	
			MSb	LSb					
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1, 2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1, 2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	—	Clear W	1	00	0001	0xxx	xxxx	Z	
COMP	f, d	Complement f	1	00	1001	dfff	ffff	Z	1, 2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1, 2
DECFSZ	f, d	Decrement f, Skip if 0	1 ⁽²⁾	00	1011	dfff	ffff		1, 2, 3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1, 2
INCFSZ	f, d	Increment f, Skip if 0	1 ⁽²⁾	00	1111	dfff	ffff		1, 2, 3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1, 2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1, 2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	—	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1, 2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1, 2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1, 2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1, 2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1, 2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1, 2
BTFSC	f, b	Bit Test f, Skip if Clear	1 ⁽²⁾	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 ⁽²⁾	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	—	Clear Watchdog Timer	1	00	0000	0110	0100	<u>TO,PD</u>	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	—	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	—	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	—	Go into Standby mode	1	00	0000	0110	0011	<u>TO,PD</u>	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Ett Assemblerprogram

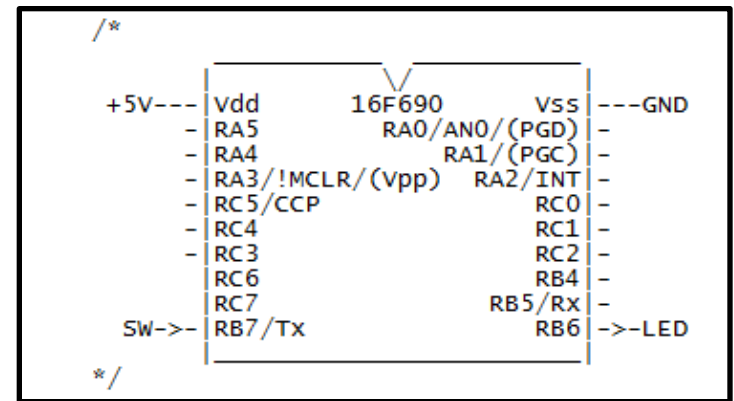
```
init
    CLRF PORTB;
    MOVLW 10111111b;
    MOVWF TRISB;
loop
    BTFSS PORTB,7;
    GOTO lampoff;
lampon
    BSF PORTB,6;
    GOTO loop;
lampoff
    BCF PORTB,6;
    GOTO loop;
end;
```



Programmet tänders och släcker lysdioden på kommando från strömbrytaren.

(Detta går naturligtvis lika bra utan PIC – men då är det ju *ingen sport!*)

Kommenterat assemblerprogram



Assemblerprogram är så kallad "spaghettiprogramering". Det blir lättare att följa programhoppnen när man ritar ut pilar.

init

```
CLRF PORTB;      0-ställ register portB
MOVLW 10111111b; hämta en konstant till arbetsregistret W
MOVWF TRISB;     kopiera konstanten till trisB registret
```

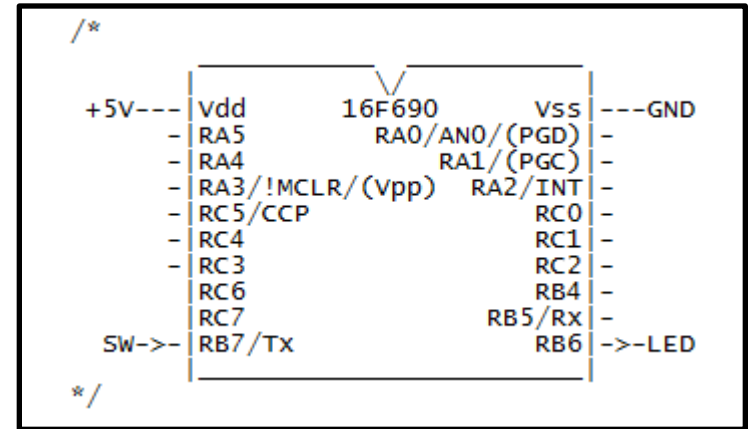
```
→ loop
  BTFSS PORTB,7; hoppa över nästa instruktion om portb.7 = 1
  GOTO lampoff;  hoppa till "lampoff"
→ lampon
  BSF PORTB,6;   1-ställ portB.6 -> tänd LED
  GOTO loop;     börja om från "loop"
→ lampoff
  BCF PORTB,6;   0-ställ portB.6 -> släcker LED
  GOTO loop;     börja om från "loop"
end;
```

C-program

```
/* onoff.c */
/* B Knudsen Cc5x */
/* C-compiler */
/* not ANSI-C */
```

```
#include "16F690.h "
#pragma config |= 0x00D4
```

```
void main( void)
{
    TRISB.6 = 0;
    PORTB.7 = 1;
    while(1)
    {
        if ( PORTB.7==1 ) PORTB.6=1;
        else PORTB.6=0;
    }
}
```



Pragma – utvidgningar av C-språket
Bitvariabler **variabel.bit**
Kompilatorn känner till namn på många register, övriga namn finns i processortypens includefil

Nedladdningsformat

Programkoden laddas ned till chippet med en kretsprogrammerare.



Det använda formatet är en textfil med op-koderna som en följd av Hex-siffror. Så här ser nedladdningskoden ut för det tidigare programexemplet.

```
:1000000001288316031307108312071483120313A6
```

```
:10001000871C0C28071406288312031307100628D0
```

```
:02400E00D400DC
```

```
:00000001FF
```

Slut på filen.

Kompilatorns ”rapport” SFR/GPR

```
RAM: 00h : -----
RAM: 20h : ==.*****
RAM: 40h : *****
RAM: 60h : *****
RAM: 80h : -----
RAM: A0h : *****
RAM: C0h : *****
RAM: E0h : *****
```

Codepage 0 has 68 word(s) : 3 %

Codepage 1 has 0 word(s) : 0 %

Program

Symbols:

- * : free location
- : predefined or pragma variable
- = : local variable(s)
- . : global variable

(Cc5x interna variabler)

Inbyggt i kompilatorn finns följande namn på register och flaggor (bitar i register):

```
char W;
char INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB;
char OPTION, TRISA, TRISB;
/* STATUS : */ bit Carry, DC, Zero_, PD, TO, PA0, PA1, PA2;
/* FSR : */ bit FSR_5, FSR_6; char PORTC, TRISC; char PCLATH, INTCON;
/* OPTION : */ bit PS0, PS1, PS2, PSA, T0SE, T0CS, INTEDG, RBPU_;
/* STATUS : */ bit Carry, DC, Zero_, PD, TO, RP0, RP1, IRP;
/* INTCON : */ bit RBIF, INTF, T0IF, RBIE, INTE, T0IE, GIE;
```

Dessa ska *inte* deklareras i programmen. Includefilerna innehåller sedan ytterligare registernamn och namn på bitar, med de beteckningar de har i processortypens officiella manual.

(Cc5x interna funktioner)

De interna funktionerna ger ”direktåtkomst” till några av PIC-processorns instruktioner:

```
btsc(Carry); // void btsc(char); - BTFSC f,b
btss(bit2); // void btss(char); - BTFSS f,b
clrwdt(); // void clrwdt(void); - CLRWDT
i = decsz(i); // char decsz(char); - DECFSZ f,d
W = incsz(i); // char incsz(char); - INCFSZ f,d
nop(); // void nop(void); - NOP
nop2(); // void nop2(void); - GOTO next address
retint(); // void retint(void); - RETFIE
W = rl(i); // char rl(char); - RLF i,d
i = rr(i); // char rr(char); - RRF i,d
sleep(); // void sleep(void); - SLEEP
skip(i); // void skip(char); - computed goto
k = swap(k); // char swap(char); - SWAPF k,d
```

`clearRAM(); // void clearRAM(void);` En intern funktion som man kan anropa för att 0-ställa allt dataminne i processorn.

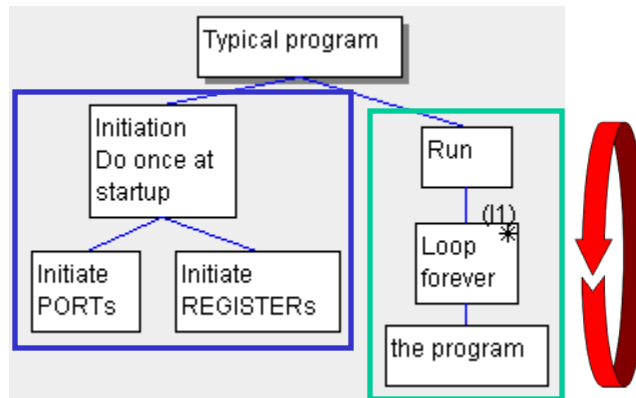
(Enkla C-satser → Assembler)

Enkla C-satser översätts i allmänhet direkt till ensamma assemblerinstruktioner. Program skrivna i assembler kan därför översättas instruktion för instruktion till ett **Cc5x** C-program.

<code>nop {};</code>	<code>NOP</code>	<code>W = f;</code>	<code>MOVF f,W</code>
<code>f = W;</code>	<code>MOVWF f</code>	<code>W = f ^ 255;</code>	<code>COMF f,W</code>
<code>W = 0;</code>	<code>CLRW</code>	<code>f = f ^ 255;</code>	<code>COMF f</code>
<code>f = 0;</code>	<code>CLRF</code>	<code>W = f + 1;</code>	<code>INCF f,W</code>
<code>W = f - W;</code>	<code>SUBWF f,W</code>	<code>f = f + 1;</code>	<code>INCF f</code>
<code>f = f - W;</code>	<code>SUBWF f</code>	<code>b = 0;</code>	<code>BCF f,b</code>
<code>W = f - 1;</code>	<code>DECF f,W</code>	<code>b = 1;</code>	<code>BSF f,b</code>
<code>f = f - 1;</code>	<code>DECF f</code>	<code>return 5;</code>	<code>RETLW 5</code>
<code>W = f W;</code>	<code>IORWF f,W</code>	<code>s1{};</code>	<code>CALL s1</code>
<code>f = f W;</code>	<code>IORWF f</code>	<code>goto X</code>	<code>GOTO X</code>
<code>W = f & W;</code>	<code>ANDWF f,W</code>	<code>W = 45;</code>	<code>MOVLW 45</code>
<code>f = f & W;</code>	<code>ANDWF f</code>	<code>W = W 23;</code>	<code>IORLW 23</code>
<code>W = f ^ W;</code>	<code>XORWF f,W</code>	<code>W = W & 53;</code>	<code>ANDLW 53</code>
<code>f = f ^ W;</code>	<code>XORWF f</code>	<code>W = W ^ 12;</code>	<code>XORLW 12</code>
<code>W = f + W;</code>	<code>ADDWF f,W</code>	<code>W = 33 + W;</code>	<code>ADDLW 33</code>
<code>f = f + W;</code>	<code>ADDWF f</code>	<code>W = 33 - W;</code>	<code>SUBLW 33</code>

Typiska programstrukturer

Ett typiskt program



```
int main()
{
    /* Initiate - PORTs */
    /* Initiate - REGISTERs */
    while (1)
        /* the program */
}
```

The screenshot shows a window titled 'C Program' with a 'Fil' menu. The code is as follows:

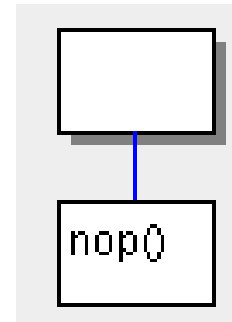
Ett typiskt program.

Först initierar man PORTAR och enheter så att de ställs in för att passa tillämpningen. Detta görs **en gång** i början av programmet.

Sedan går programmet in i en **evighetsloop** – reagerar på insignaler och levererar utsignaler varje varv i loopen.

Programmet avslutas när matningspänningen slås av.

Engångsprogram?



- C-program:

```
void main( void)
{
    nop(); /* to do something once */
}
```

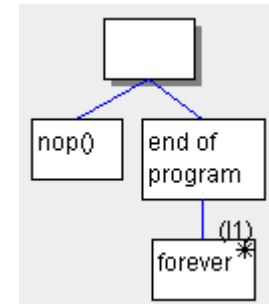
- Översatt till assembler:

```
→ main
    NOP
    SLEEP
    GOTO main
    END
```

```
; nop(); /* to do something once */
; }
```

Engångsprogram går inte, kompilatorn lägger ut **SLEEP**, så processorn går in i strömsparläge. Det gör även IO-enheterna.

Engångsprogram?



- C-program:

```
void main( void)
{
    nop(); /* something once */
    while(1);
}
```

- Översatt till assembler:

```
main
; nop(); /* something once */
NOP
; while(1) ;
→ m001 GOTO m001
END
```

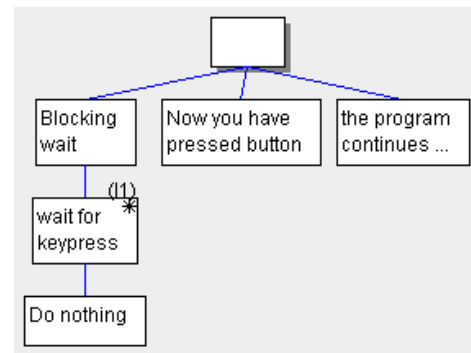
Så här ser ett engångsprogram ut som *inte* tvingar fram **SLEEP**, strömsparläget.

Vänta på en knapptryckning?



PORTB bit 0
blir 1 när man
trycker

Många gånger har processorn inte så mycket att göra, då kan man använda **blockerande kod**.



- Vänta på en knapptryckning, blockerande:

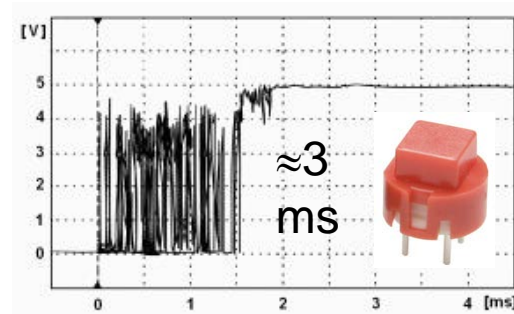
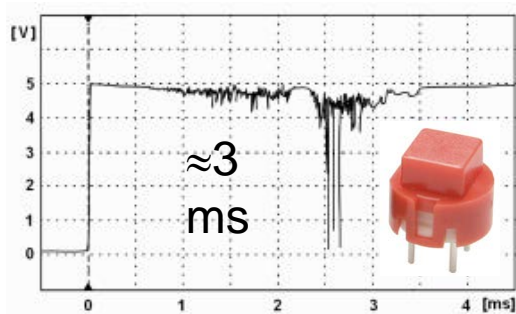
```
while (PORTB & 0x01 == 0) /* do nothing */ ;  
/* OK, now you have pressed the button ... */
```

- Eller enklare – PIC-processorn har ju bitvariabler:

```
while (!PORTB.0) /* do nothing */ ;  
/* OK, now you have pressed the button ... */
```

Kontaktstudsar!

När man trycker, eller släpper, en mekanisk kontakt så studsar den ett tag mot kontaktytan innan den lägger sig till ro. PIC-processorn är så snabb att den kan uppfatta varje studs som en "egen" kontakt-tryckning!



Om en kontakt studsar mycket eller lite syns inte på utsidan!

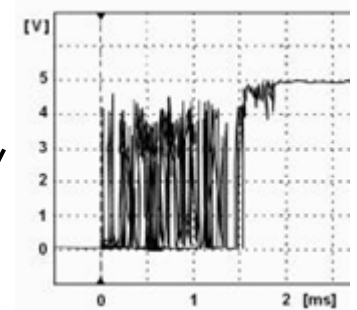


Togglar en LED ON/OFF



Rena rama slumpgeneratorn, vad som helst kan **hända/inte hända** när man trycker på knappen!

```
void main( void)
{
  TRISB = 0b10111111; /* RB7 in, RB6 out */
  while(1)
  {
    while( !PORTB.7 ) ; /* wait key pressed */
    PORTB.6 = !PORTB.6; /* toggle led */
    while( PORTB.7 ) ; /* wait for key released */
  }
}
```





- **Inte som tänkt, varannan gång – utan en slumpgenerator!**



Togglar en LED ON/OFF



Vänta ut kontaktstudsarna. En kontakt kan studsas både när man **trycker ned** den och när man **släpper upp** den!

```
void main( void)
{
    TRISB = 0b10111111; /* RB7 in, RB6 out */
    while(1)
    {
        while( !PORTB.7 ) ; /* wait key pressed */
        PORTB.6 = !PORTB.6; /* toggle led */
        delay(5);  Vänta ut kontaktstudsarna (>5ms)
        while( PORTB.7 ) ; /* wait for key released */
        delay(5);  Vänta ut kontaktstudsarna (>5ms)
    }
}
```

• Nu fungerar det!

delay() funktionen

C-funktioner

```
void delay(char);
```

• Funktionsdeklaration (prototyp) före main()

```
void main( void)
```

```
{
```

```
    TRISB = 0b10111111; /* RB7 in, RB6 out */
```

```
    while(1)
```

```
    {
```

```
        while( !PORTB.7 ) ; /* wait key pressed */
```

```
        PORTB.6 = !PORTB.6; /* toggle led */
```

```
        delay(5); • Funktionsanrop
```

```
        while( PORTB.7 ) ; /* wait for key released */
```

```
        delay(5); • Funktionsanrop
```

```
    }
```

```
}
```

• Placera funktionsdefinitionerna efter main() i en och samma fil.

delay() funktionen

- Placera funktionsdefinitionerna efter main() i en och samma fil.

```
/* Delays a multiple of 1 milliseconds at 4 MHz */  
/* (16F690 internal clock) using the TMR0 timer */
```

```
void delay( char millisec )
```

```
{
```

```
    OPTION = 2; /* prescaler divide by 8 */
```

```
    do
```

```
    {
```

```
        TMR0 = 0;
```

```
        while ( TMR0 < 125) /* 125 * 8 = 1000 */ ;
```

```
    } while ( -- millisec > 0);
```

```
}
```

millisec
antal varv

1000 μ s

Det är den eftertestade slingan som är den iterationsmetod som bäst passar PIC-processorn.

```
do  
{  
    --- ;  
} while(---);
```


TIMER0

TIMER0 är en inbyggd 8-bitars modulo 256-räknare som kan läsas/skrivas från programmen. När timern “räknat runt” sätts biten **TOIF**.

Om biten **TOCS** i **OPTION** -registret är "0" så är det processorns klockcykler som räknas. Om biten **TOCS** är "1" räknas pulsflanker från pinnen **TOCKI**.

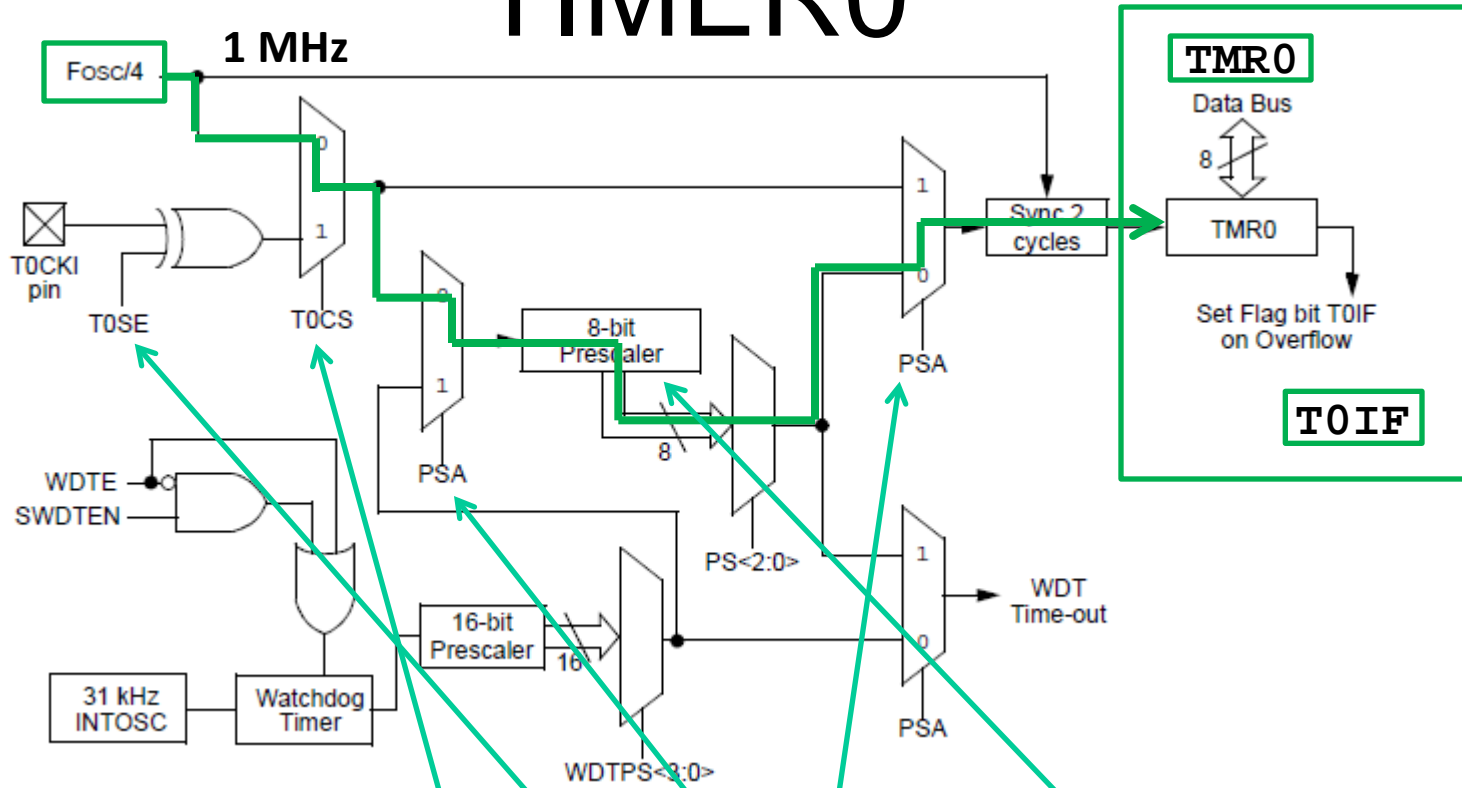
PS2	PS1	PS0	Prescaler
000			1:2
001			1:4
010			1:8
011			1:16
100			1:32
101			1:64
110			1:128
111			1:256

Med biten **PSA=0** kan man koppla in en **prescaler**, en frekvensdelare.

Med den inkopplad räknas bara en bråkdel av de inkommande pulserna, så att timern räknar långsammare. Med bitarna **PS2 PS1 PS0** ställer man in prescalerns delningstal.

```
TMRO=0;      /* reset timer0 */  
time=TMRO; /* store timer0 value in char variable time */  
TMRO=17;     /* preset timer0 to 17 */
```

TIMER0



REGISTER 5-1: OPTION_REG: OPTION REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
\overline{RABPU}	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

C-funktioner sammanfattning

- Funktionsdeklarationer före **main()**.
- Anrop från **main()** eller från andra funktioner.
- Funktionsdefinitioner efter **main()**, i samma fil.

Det gäller oftast så lite programkod att allt kan vara i en och samma fil. Funktionerna blir ofta skräddarsydda för tillämpningen och processorn, därför kan det vara onödigt att lagra dem i något ”funktionsbibliotek”.

Vänta på knapptryckningar?

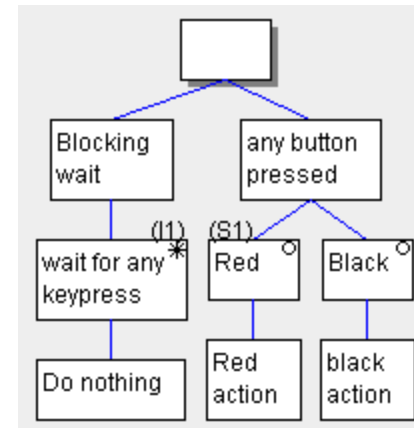


PORTB bit 0
blir 1 när man
trycker



PORTB bit 1
blir 1 när man
trycker

Två knappar, **blockerande kod.**



eller

```
while(!PORTB.0 || !PORTB.1) /* do nothing */ ;
```

```
/* now one or both buttons are pressed */  
if(PORTB.0) /* action for red button */ ;  
if(PORTB.1) /* action for black button */ ;
```

Reagera på knapptryckningar?



PORTB bit 0
blir 1 när man
trycker



PORTB bit 1
blir 1 när man
trycker

Två knappar, **icke blockerande kod**

bit flagbit;

```
while(1) /* main programloop */
```

```
{
```

```
/* examine button status */
```

```
if(PORTB.0) /* direct action for red button */ ;
```

```
if(PORTB.1) flagbit = 1; else flagbit = 0;
```

```
/* . . . */
```

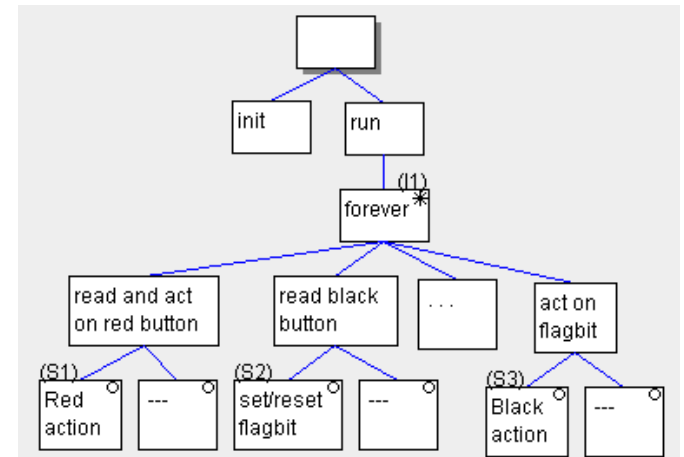
```
/* later, act on the flagbit */
```

```
if(flagbit) /* action for black button */ ;
```

```
}
```

• Kontaktstudsar?

Man kan reagera direkt på knappstatus *eller* föra informationen vidare med en bitvariabel, en flagg-bit.



Reagera på knapptryckningar?



PORTB bit 0
blir 1 när man
trycker



PORTB bit 1
blir 1 när man
trycker

Två knappar, **icke blockerande kod**

bit flagbit;

```
while(1) /* main programloop */
```

```
{
```

```
/* examine button status */
```

```
if(PORTB.0) /* direct action for red button */ ;
```

```
if(PORTB.1) flagbit = 1; else flagbit = 0;
```

```
/* . . . */
```

```
/* later, act on the flagbit */
```

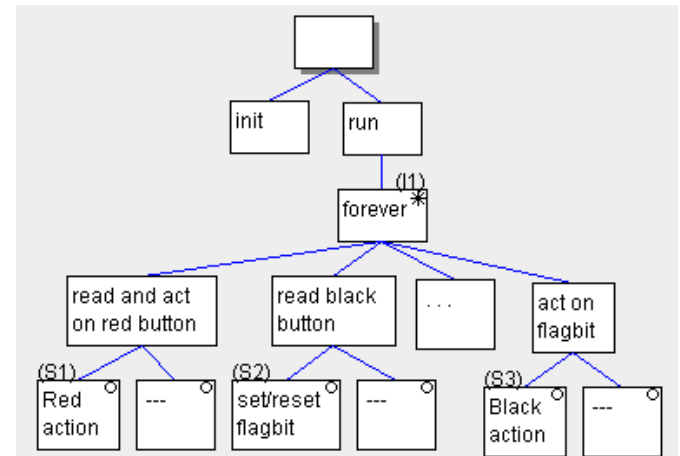
```
if(flagbit) /* action for black button */ ;
```

```
}
```

```
delay(5);
```



Vänta ut (>5ms) kontaktstudsarna
innan nästa varv i main-loopen



Checkbox eller Radiobutton?

Checkbox (många alternativ):

```
if(a)b; if(c)d; if(e)f; . . .
```

What did you like about the Site?

Cool Layout ?

Easy to Navigate

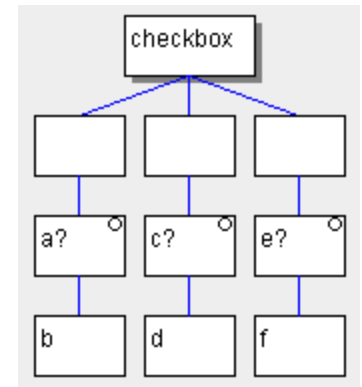
Great Contents

What did you hate about the site?

Awful Layout ?

Difficult to Navigate

Lousy Contents

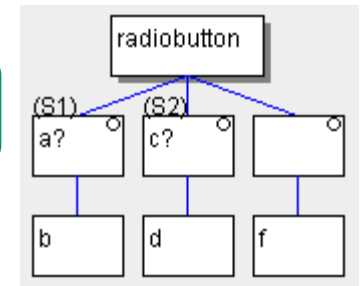


Radio Button (endast ett):

```
if(a)b; else if(c)d; ... else f;
```

Your Location:

North East North West South East South West Midlands



Radiobutton ...

Att välja endast **ett alternativ** bland flera ...

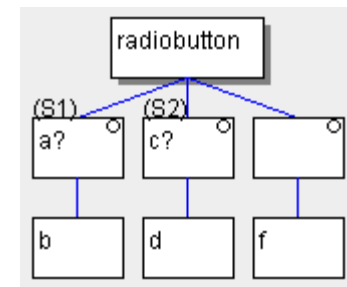
Your Location:

North East North West South East South West Midlands

```
if(a) b;
```

```
else if(c) d;
```

```
else f;
```



Eller med C-språkets **switch-case** uttryck ...

C-språkets **switch** – **case** uttryck

Tips! Observera att *B Knudsen* kompilatorn lägger ut effektivare kod för

- **switch()** – **case**

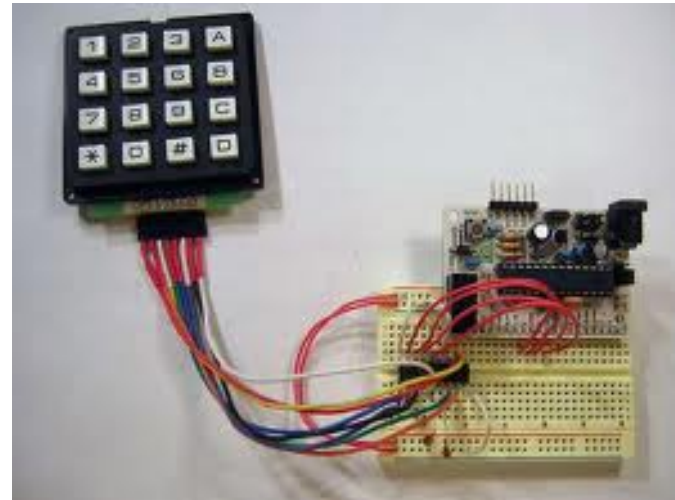
än för

- **if()** – **else if()** – **else**

så använd alltid switch-satsen!

C's switch – case

```
switch(d) {  
case 0x00 : k='1'; break;  
case 0x01 : k='2'; break;  
case 0x02 : k='3'; break;  
case 0x04 : k='4'; break;  
case 0x05 : k='5'; break;  
case 0x06 : k='6'; break;  
case 0x08 : k='7'; break;  
case 0x09 : k='8'; break;  
case 0x0A : k='9'; break;  
case 0x0C : k='*'; break;  
case 0x0D : k='0'; break;  
case 0x0E : k='#'; break;  
/* 0x03,0x07,0x0B,0x0F */  
default  : k='  ';  
}
```



Omkodning. Tangentbord avger oftast en helt annan kod **d** än vad som är ingraverat på tangenten **k** !

Ex. Smidig meny-hantering

```
switch( choice )
{
    case 'Y' : /* Yes */
    case 'y' : /* yes */
    case 'J' : /* Ja */
    case 'j' : /* ja */
        printf( "As you wish" );
        break;
    case 'N' : /* No Nej */
    case 'n' : /* no nej */
        printf( "Ok. You don't need to" );
        break;
    default :
        printf( "Wrong answer, Y/y/J/j/N/n" );
}
```

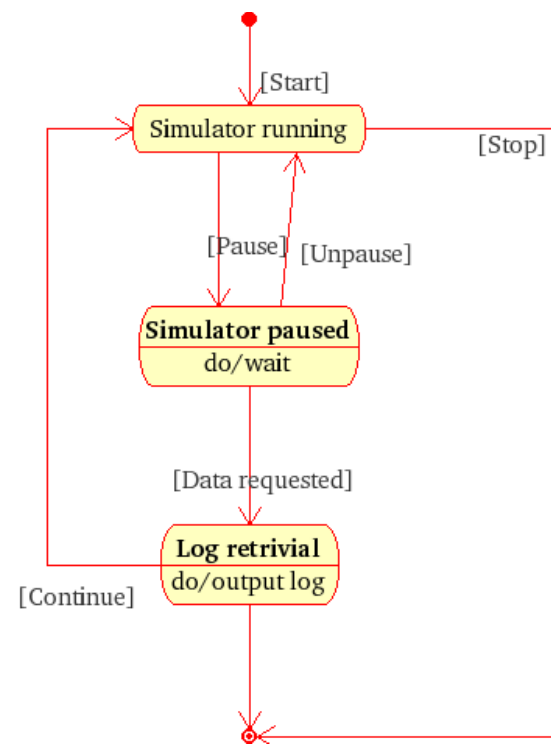
Gruppera alternativen

Default, för alla ospecificerade alternativ

Programmering efter tillståndsdigram

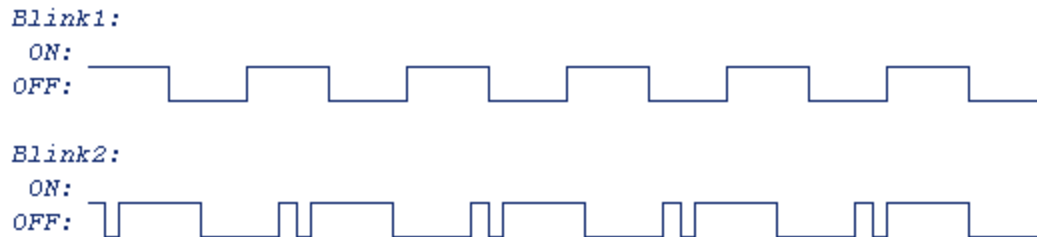
En mycket vanlig teknik vid programmering av inbyggnadsprocessorer är att använda ”tillstånd” och ”tillståndsdigram”.

Idén är lånad från **Digital Designs** ”automater”.



Ex. UML-tillståndsdigram

Flera saker samtidigt?



```
/* Blink1: 1s ON - 1s OFF */
```

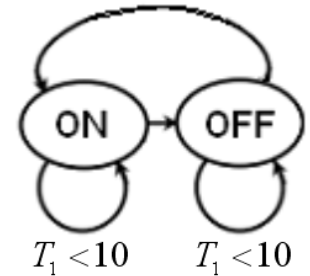
```
/* Blink2: 0,2s ON - 0,2s OFF - 1s ON - 1s OFF */
```

Först en lysdiod ...

$T_1 = 0 \Leftarrow T_1 = 10$

```
while(1)
{
    /* Blink1: 1s ON - 1s OFF */
    switch(State1)
    {
        case 0:
            PORTB_copy.6=1;      /* Blink1 = ON */
            Time1++;
            if( Time1 == 10 ) { State1 = 1; Time1 = 0; }
            break;
        case 1:
            PORTB_copy.6=0;      /* Blink1 = OFF */
            Time1++;
            if( Time1 == 10 ) { State1 = 0; Time1 = 0; }
    }
    PORTB = PORTB_copy;
    delay10(10); /* 0,1 sec delay each lap */
}
```

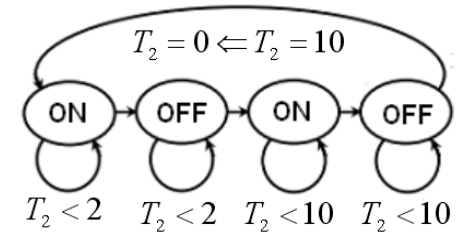
Blink1:
ON:
OFF:



Sedan en annan lysdiod ...

```
while(1)
{
  /* Blink2:  0,2s ON - 0,2s OFF - 1s ON - 1s OFF */
  switch(State2){
    case 0:
      PORTB_copy.5 = 1; Time2++; /* Blink2 ON */
      if( Time2 == 2 ) { State2 = 1; Time2 = 0; }
      break;
    case 1:
      PORTB_copy.5 = 0; Time2++; /* Blink2 OFF */
      if( Time2 == 2 ) { State2 = 2; Time2 = 0; }
      break;
    case 2:
      PORTB_copy.5 = 1; Time2++; /* Blink2 ON */
      if( Time2 == 10 ) { State2 = 3; Time2 = 0; }
      break;
    case 3:
      PORTB_copy.5 = 0; Time2++; /* Blink2 OFF */
      if( Time2 == 10 ) { State2 = 0; Time2 = 0; }
  }
  PORTB=PORTB_copy;
  delay10(10); /* 0,1 sek delay */
}
```

Blink2:



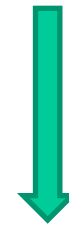
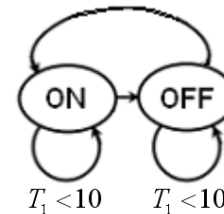
Varför inte båda?



while(1)

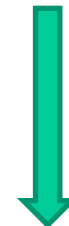
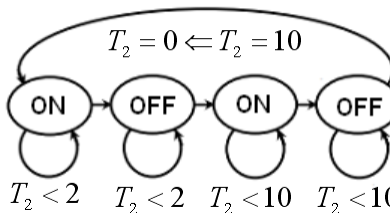
```
{  
  /* Blink1: 1s ON - 1s OFF */  
  switch(State1)  
  {  
    case 0: ... ; break;  
    case 1: ... ;  
  }  
}
```

$$T_1 = 0 \Leftarrow T_1 = 10$$



snabbt
10 μ s

```
/* Blink2: 0,2s ON - 0,2s OFF - 1s ON - 1s OFF */  
switch(State2)  
{  
  case 0: ... ; break;  
  case 1: ... ; break;  
  case 2: ... ; break;  
  case 3: ... ;  
}
```



snabbt
10 μ s

```
PORTB = PORTB_copy;  
delay10(10); /* 0,1 sek delay */
```

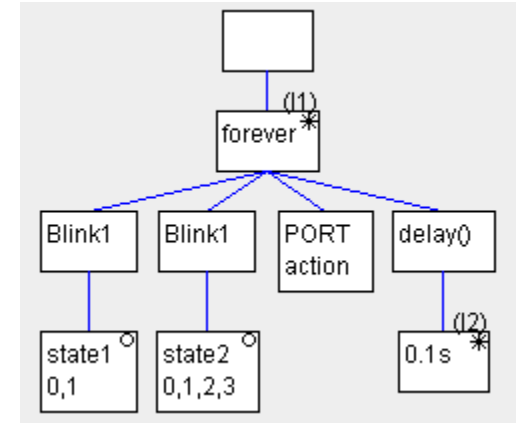


långsamt 0.1 s



State machine

Genom att programmera ”statemachines” (jämför med Digital Design) kan man få det att se ut som om processorn klarar att utföra **många saker samtidigt**. Man kan prova ut varje sak för sig, och oftast fungerar sedan hela kombinationen som tänkt.



WARNING! Det finns ett ”lömskt” så kallat **RMW**-problem.
TIPS, LÖSNING: Ändra bitar i en variabel **PORT_copy** i stället för direkt på porten **PORT**. Kopiera sedan hela denna variabel till porten, **PORT=PORT_copy;**
Mer om detta senare i kursen ...

William Sandqvist william@kth.se