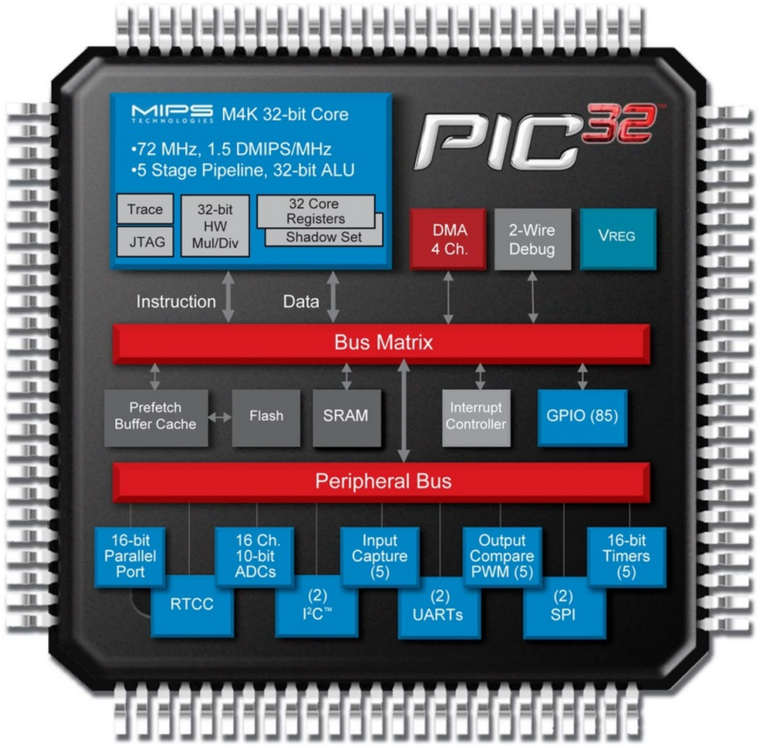
Why use a small 8-bit processor when there are cheap powerful 32-bit?



William Sandqvist william@kth.se

8-bit processor close to the sensor?

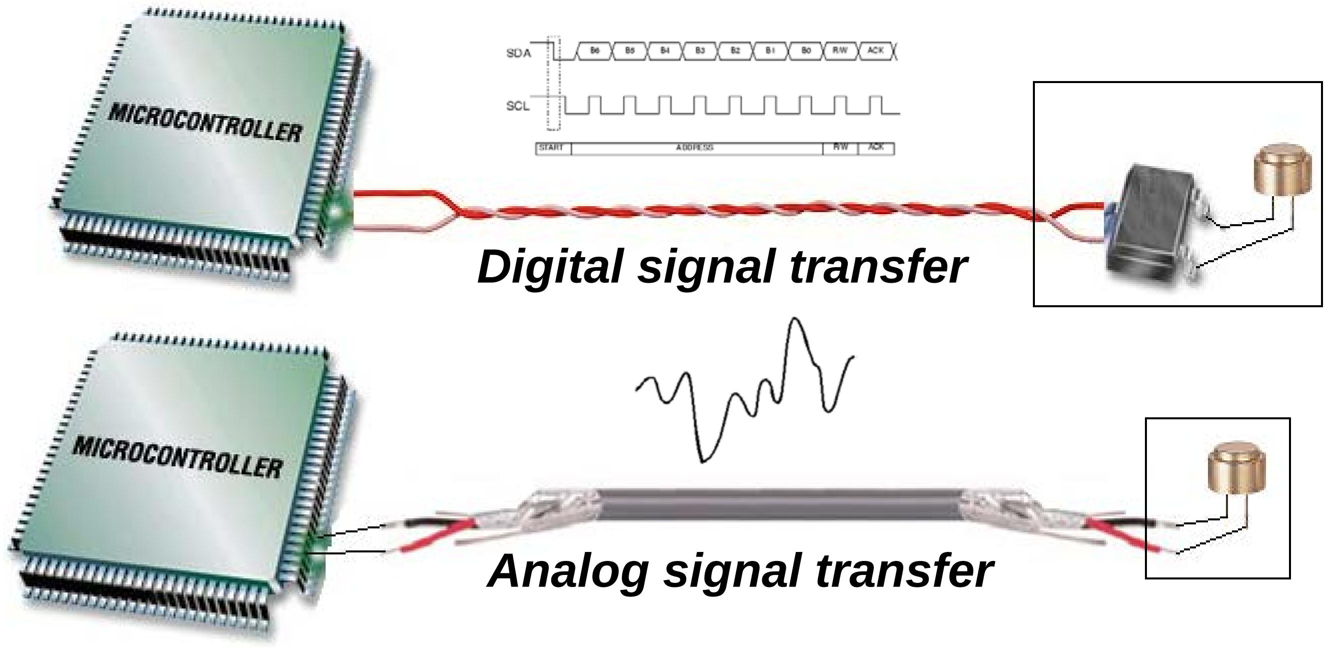
- A simple sensor often has a weak output signal. It may need to be connected with an **expensive cable**.

- An expensive sensors with "integrated electronics" can get by with a **simple cable**.

The cost of both options can very well end up to be the same!

Thus smart to embedd an 8 bit processor inside the sensor!

8-bit processor as smart cable?

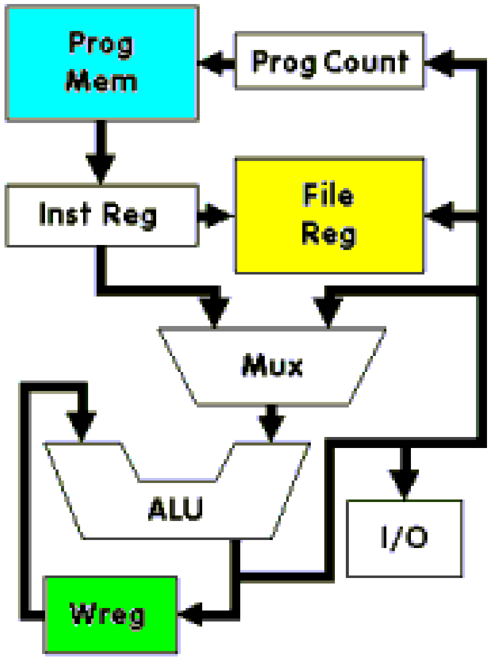


- Cheap cable
- Cheap processor

- Expensive cable
- No processor

How many 8 bit processors can you get for the cost of a meter cable? The processor as cable replacement!

PIC 8-bit processor

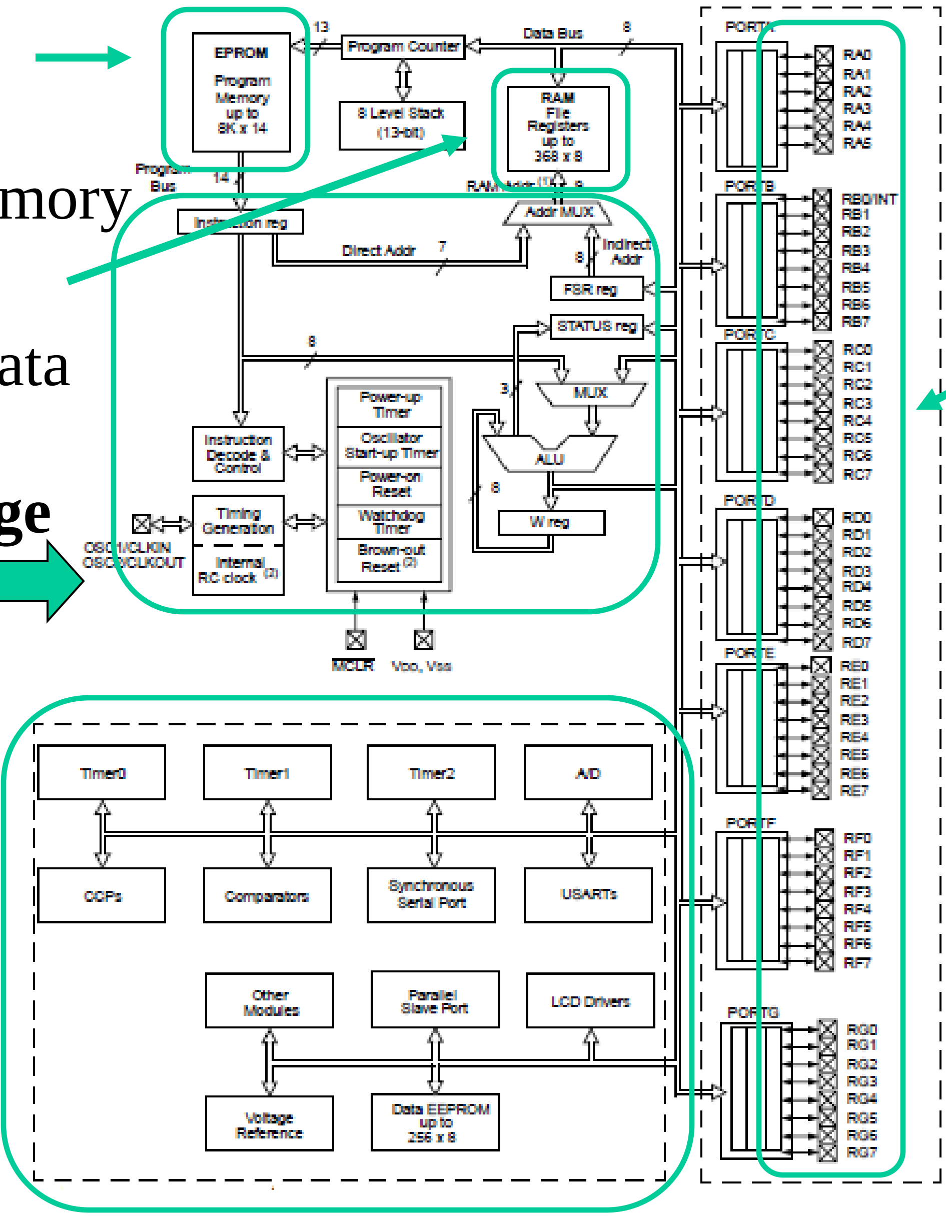


PIC (**P**eripheral **I**nterface **C**omputer) are inexpensive computer circuits with "all in one".

- Different amount of program memory
- different amount of data memory

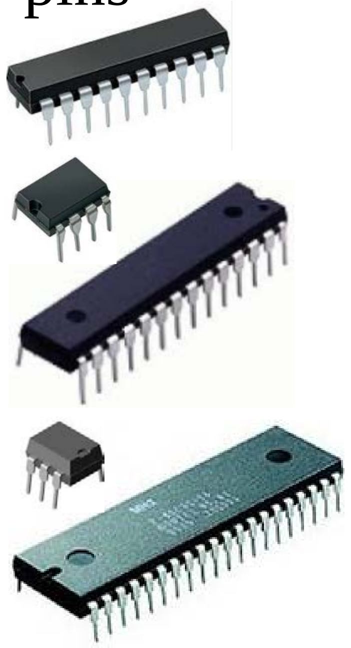
PIC Midrange processor

- Different combi-nations Of IO-units



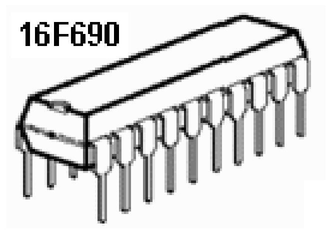
63 different typs of Midrange **PIC** processors!

- Different number of pins



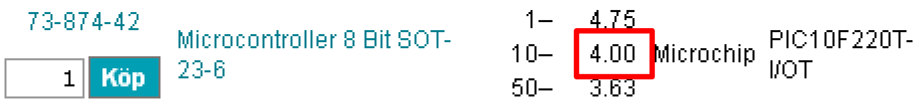
The business idea - buy only as much as you need

Develop your application on a processor with "little of everything".



To the finished product then use just exactly how much you need.

ELFA’s *cheapest* PIC-processor



73-874-42

1 Köp

Microcontroller 8 Bit SOT-23-6

1–	4.75	Microchip	PIC10F220T-I/OT
10–	4.00		
50–	3.63		

4 kr each if you buy 10 …

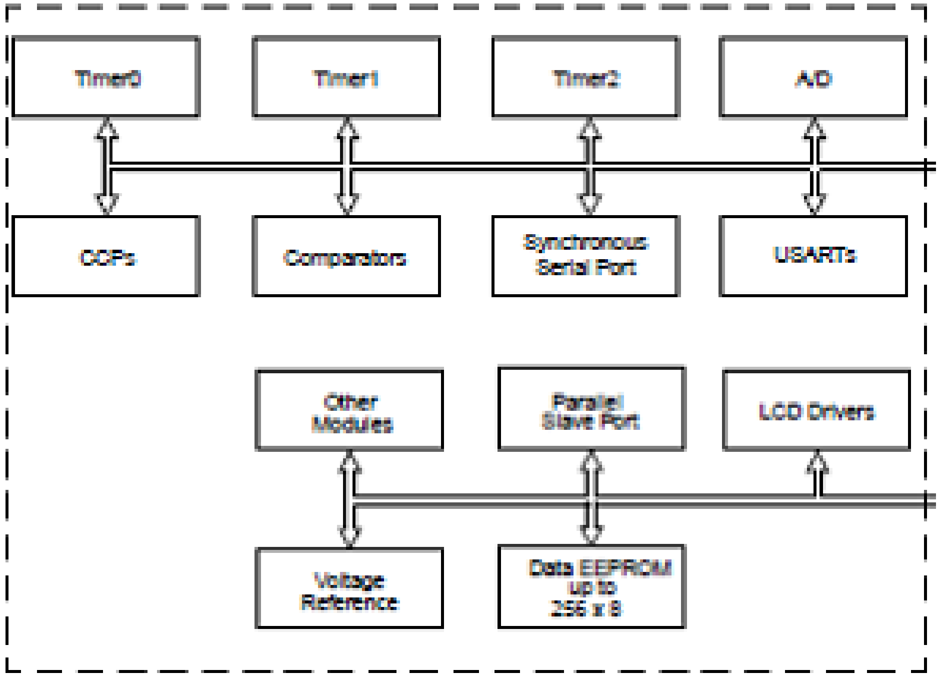
PIC10F220T-I/OT
Can be compiled with
Cc5x – includefile exist

Programmemory: 384 words
RAM-memory: 16 Byte
8 bit AD-converter 2 channels
Internal oscillator 4 MHz
TIMER0
Voltage 2…5,5 V
Typical current consumption: 175µA

When computing power is so cheap there opens up completely new possibilities…

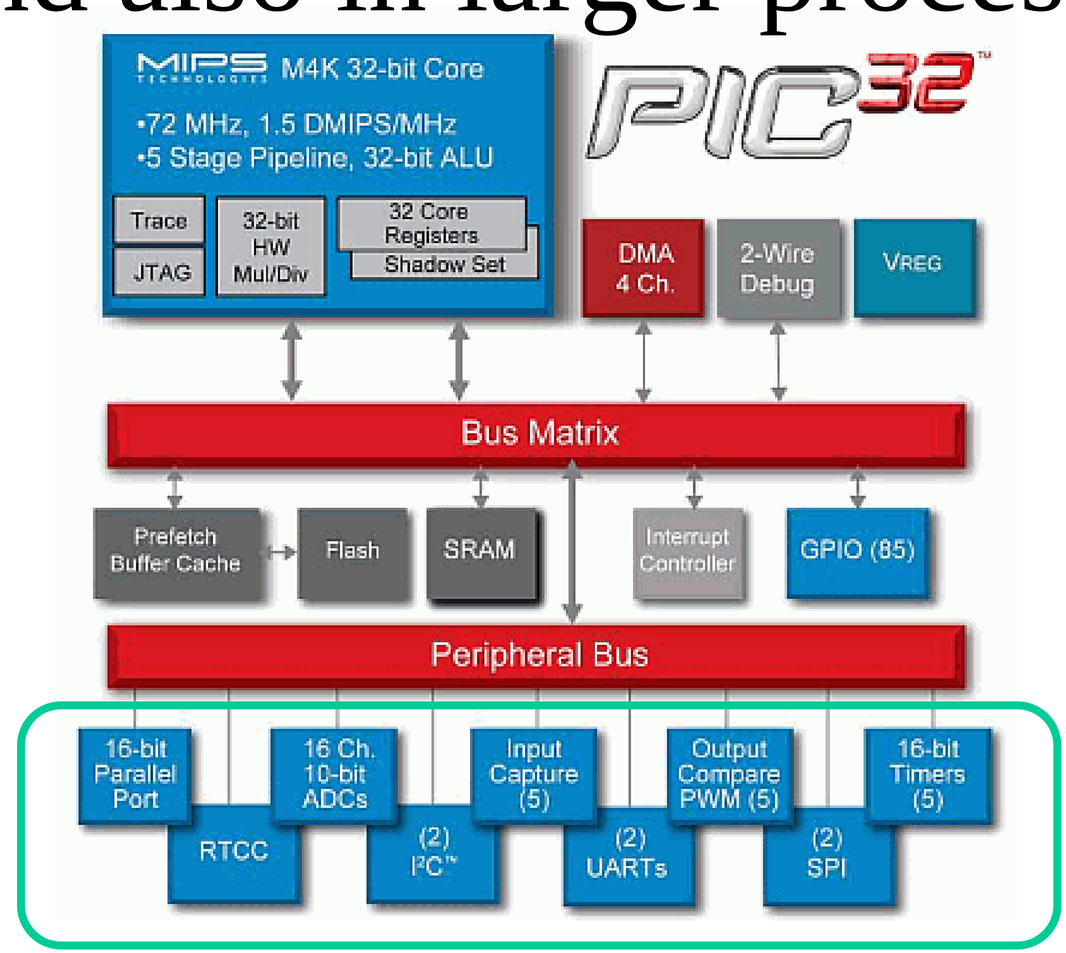
This is one reason why it might be good idea to learn PIC processors!

The built in IO devices increases 8-bit processors' **performance**



IO ports and IO bits, timers, Capture/Compare/PWM, Analog comparators, ADC, Serial ports, voltage references, data EEPROM, etc.

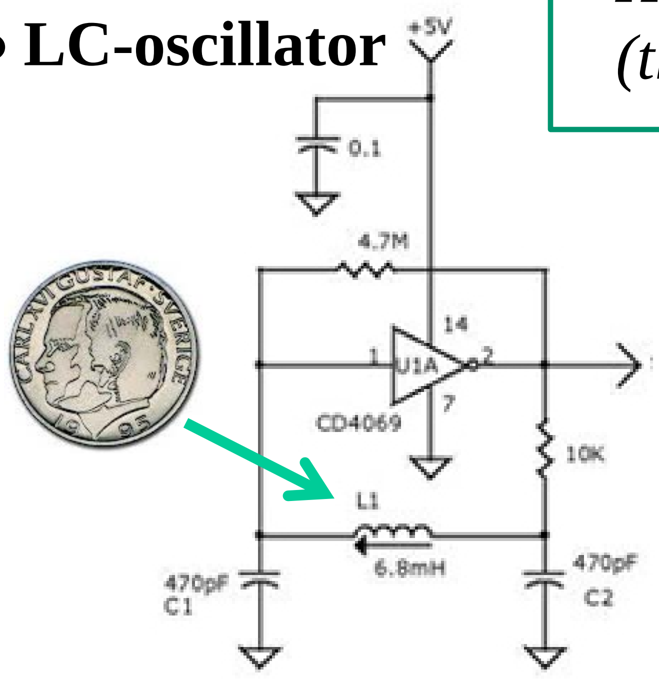
The same IO devices can then be found also in larger processors



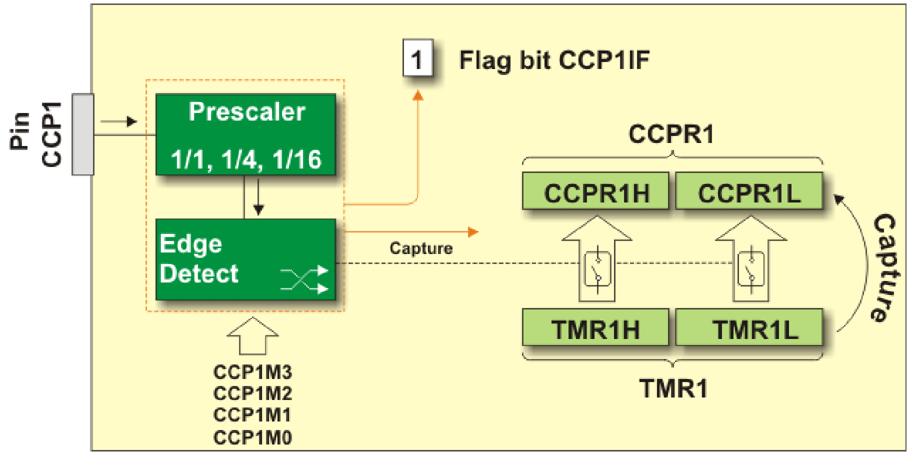
The course is all about connecting electronics to the IO devices

How to indicate that a coin is nearby (the coil)?

- LC-oscillator



- CCP-unit



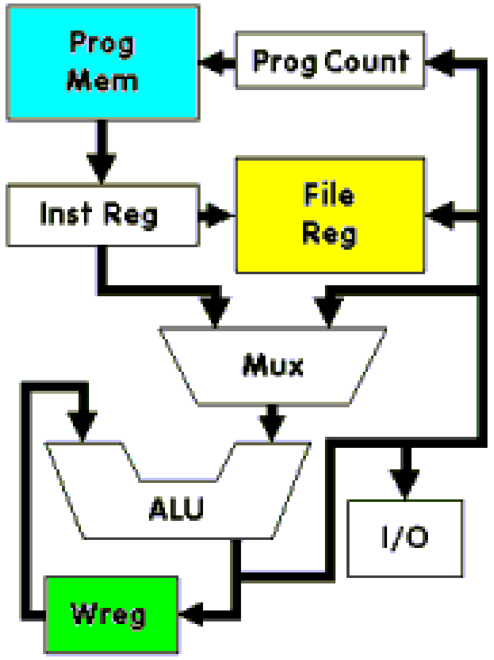
Circuit Theory and PIC processor!

William Sandqvist william@kth.se

You will, for example, get to know how an inductive sensor works…

PIC16F690

PIC 8-bit processor

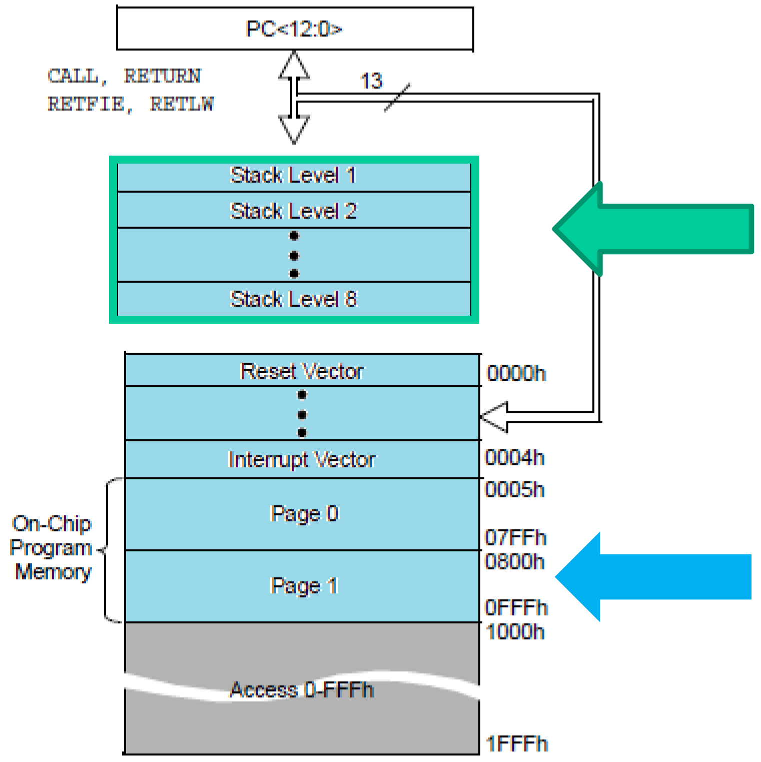


PIC (**P**eripheral **I**nterface **C**omputer) are inexpensive computer circuits with "all in one".

Prog Mem. Program memory.

File Reg. Data memory and special registers. The special register are connected to IO, for example the chip pins.

Program memory



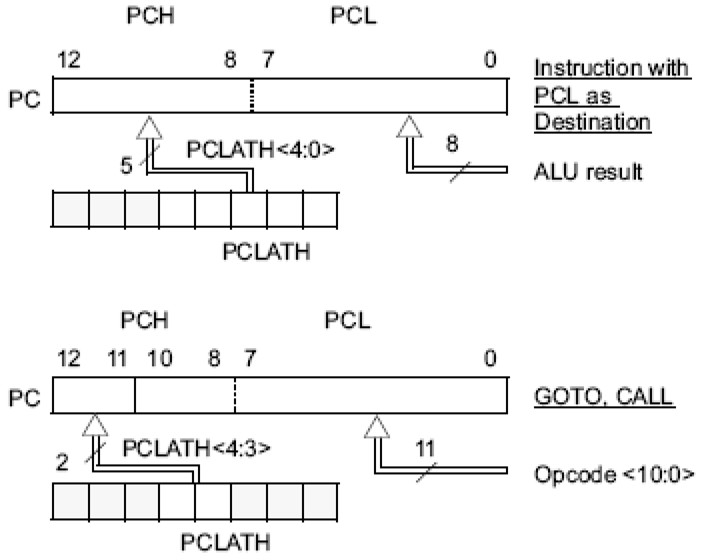
Stack
only for return adresses (8), not for parameters.

Program memory.
PIC16F690 has 7 kByte FLASH.
4096 word a’ 14 bit.

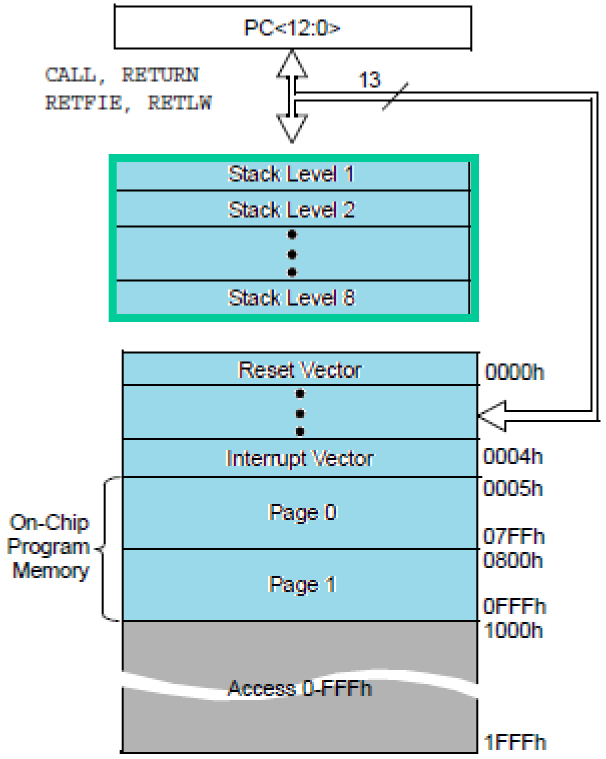
16F690 Program memory

PIC-processor **GOTO** and **CALL** -instructions can directly reach addresses within **2 k** (opcode has **11** addressbits).

16F690 has **4 k** program memory, so one has to choose new ”page” in the program-memory.



The division in pages, is an outdated architecture.



Code pages

PIC processors have the program memory divided into ”code pages"? (0, 1, 2, 3), about 2048 instructions. The compiler **Cc5x** begins to put code on page 0 and gives error when this page is not enough. Should this occur you write there instructions? `#pragma codepage 1`, then further instructions end up on the next page (and so on code page 2 if necessary).

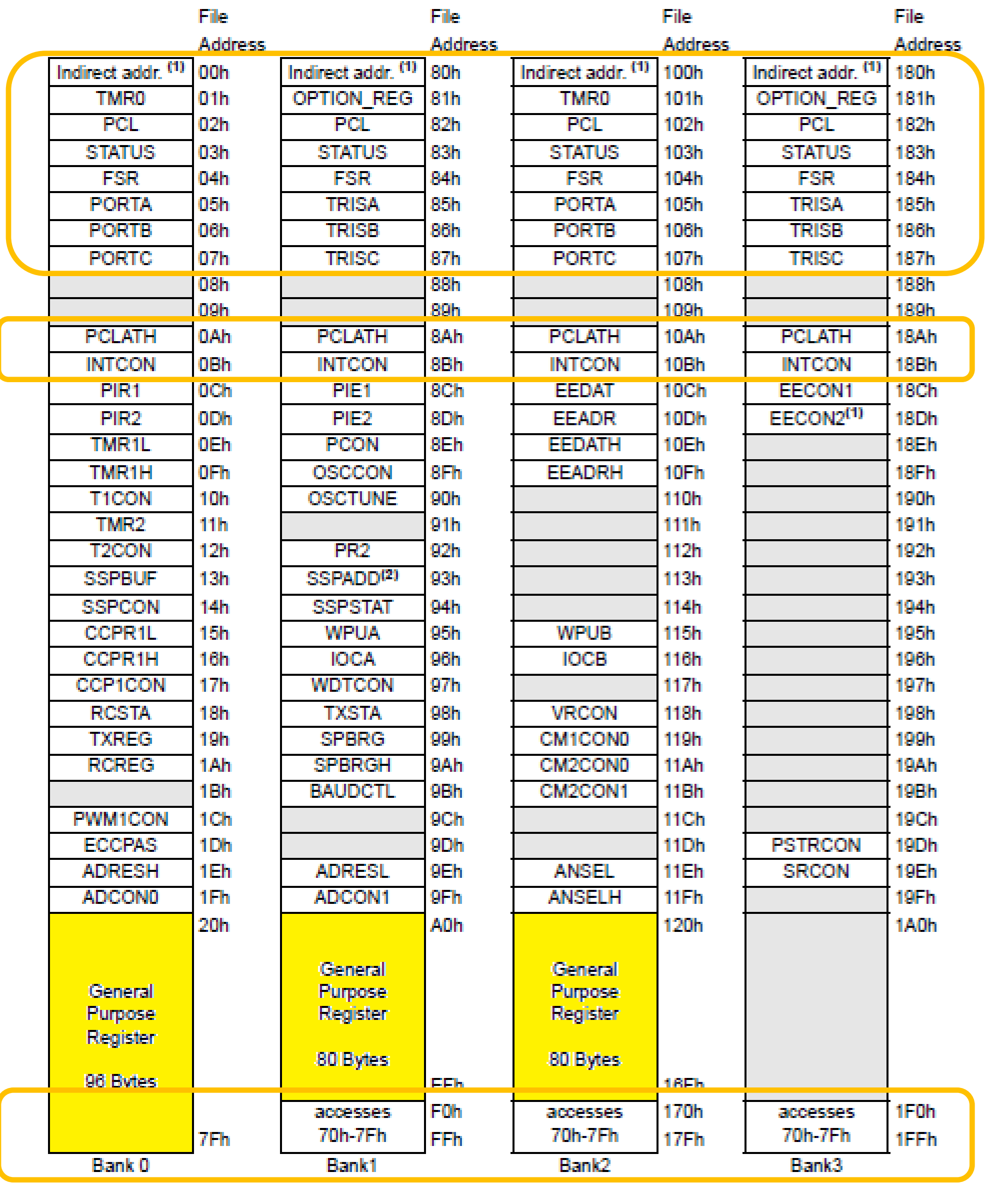
To get compact code a thorough ”page planning ” is needed, something that one hardly cares about during prototype development.

Data memory register File

PIC processor data memory is the Register File. It consists of SFR, special function registers, and the GPR General-purpose registers which are the actual data memory.

SFR registers are connected to the processor IO.

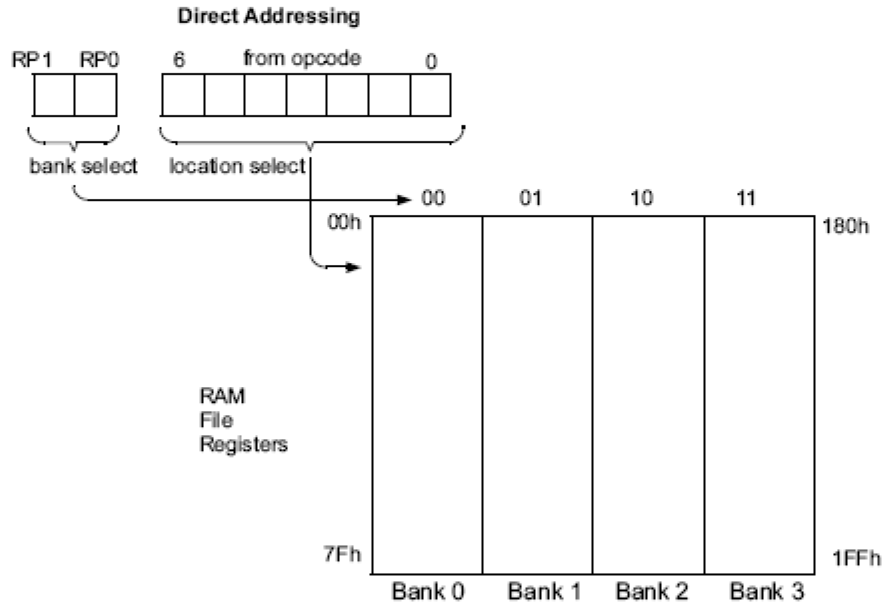
Mapped RAM, same register is found in all banks - you do not have to change rambank!



	File Address		File Address		File Address		File Address
Indirect addr. [1]	00h	Indirect addr. [1]	80h	Indirect addr. [1]	100h	Indirect addr. [1]	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	PORTA	105h	TRISA	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	PORTC	107h	TRISC	187h
	08h		88h		108h		188h
	09h		89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2[1]	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh		18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh		18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h		91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD[2]	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUA	95h	WPUB	115h		195h
CCPR1H	16h	IOCA	96h	IOCB	116h		196h
CCP1CON	17h	WDTCON	97h		117h		197h
RCSTA	18h	TXSTA	98h	VRCON	118h		198h
TXREG	19h	SPBRG	99h	CM1CON0	119h		199h
RCREG	1Ah	SPBRGH	9Ah	CM2CON0	11Ah		19Ah
	1Bh	BAUDCTL	9Bh	CM2CON1	11Bh		19Bh
PWM1CON	1Ch		9Ch		11Ch		19Ch
ECCPAS	1Dh		9Dh		11Dh	PSTRCON	19Dh
ADRESH	1Eh	ADRESL	9Eh	ANSEL	11Eh	SRCON	19Eh
ADCON0	1Fh	ADCON1	9Fh	ANSELH	11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h		1A0h
			EFh		16Fh		
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h-7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank1		Bank2		Bank3	

RP1 and **RP0**

One chooses bank with the bits **RP1** and **RP0** in **STATUS** register



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

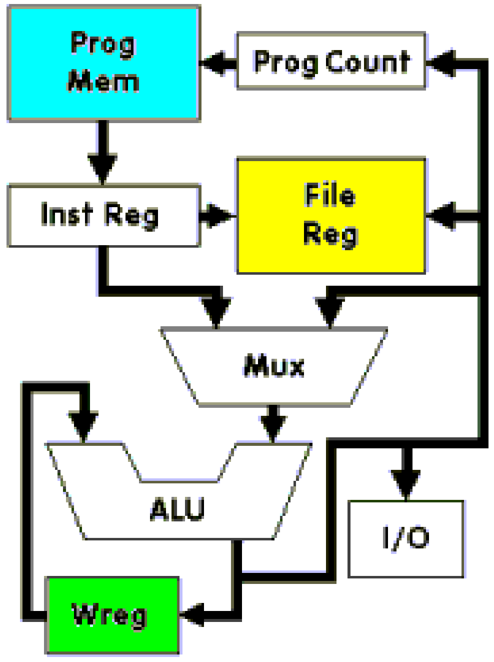
The division of data memory in RAM banks is an outdated architecture.

The compiler can choose for us!

The PIC processor's register area (RAM) is divided into "ram banks" (0, 1, 2, 3). **Cc5x** begins to fill rambank 0. You can change rambank with instruction `#pragma Rambank 1` and then all variables that are declared are placed in the next rambank (rambank 1). Some memory cells are found in the **same place** in all ram banks, known as mapped RAM. You can choose to place variables as "mapped ram" (as long as there is space) with the instruction `#pragma rambank -`.

Best use of RAM banks requires a lot of planning, something one hardly cares about during prototype development.

PC, IR, ALU, W-register



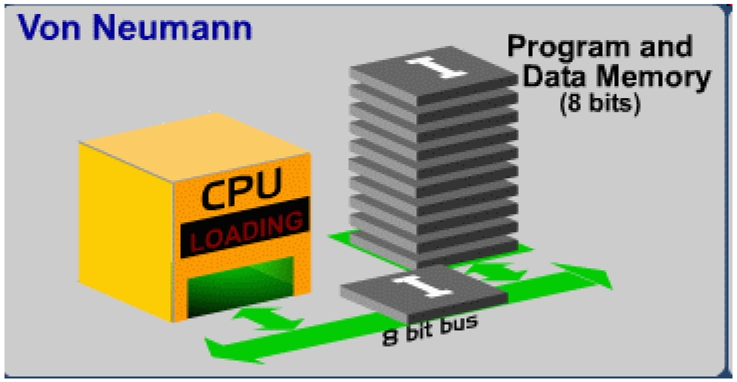
Prog Counter, PC. Programcounter register points to where in program memory the current instruction is. It is incremented automatically after each executed instruction.

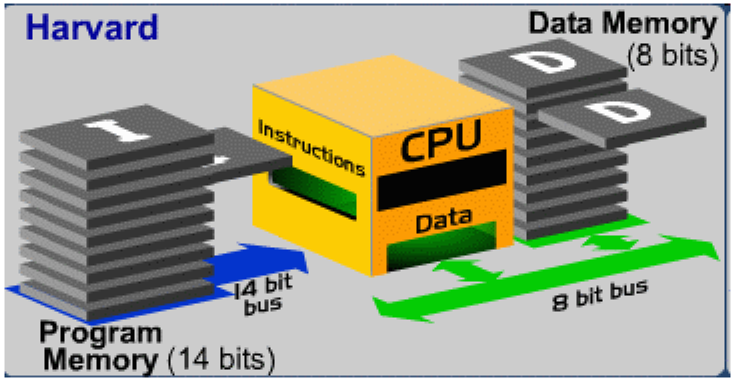
Inst Register, IR. Instruction register holds the code for the current instruction.

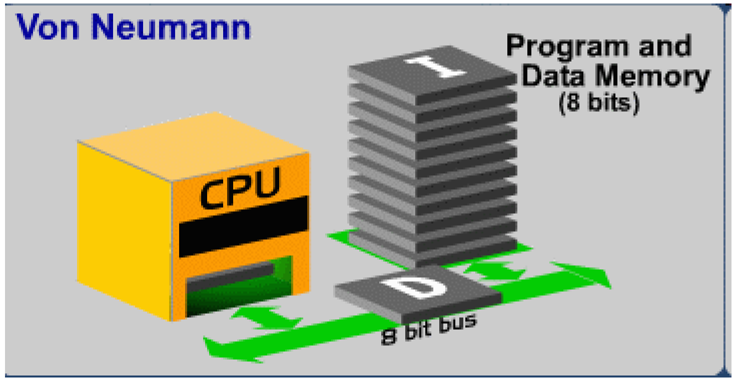
A**LU**. Arithmetisc Logic Unit handles the calculations.

The vast majority of operations are performed through the working register, W-reg. This is the PIC processor "wasp waist".

Harvard vs Von Neumann







- **Von Neumann** architecture have a common bus for instructions and data.
- **Harvard** architecture has *different* busses for instruc-tions and data.

Harvard is (twice) faster …

CISC vs RISC

- **CISC** (Complex Instruction Set Computer)
  Eg. Intel PC, has **700** instructions.
- **RISC** (Reduced Instruction Set Computer)
  Eg. Microchip PIC, has **33** instructions.

These concepts are now obsolete. Intel processors are still classified as CISC - but they have advanced architecture that utilizes all the best of RISC…

KIA’s factory in Slovenia

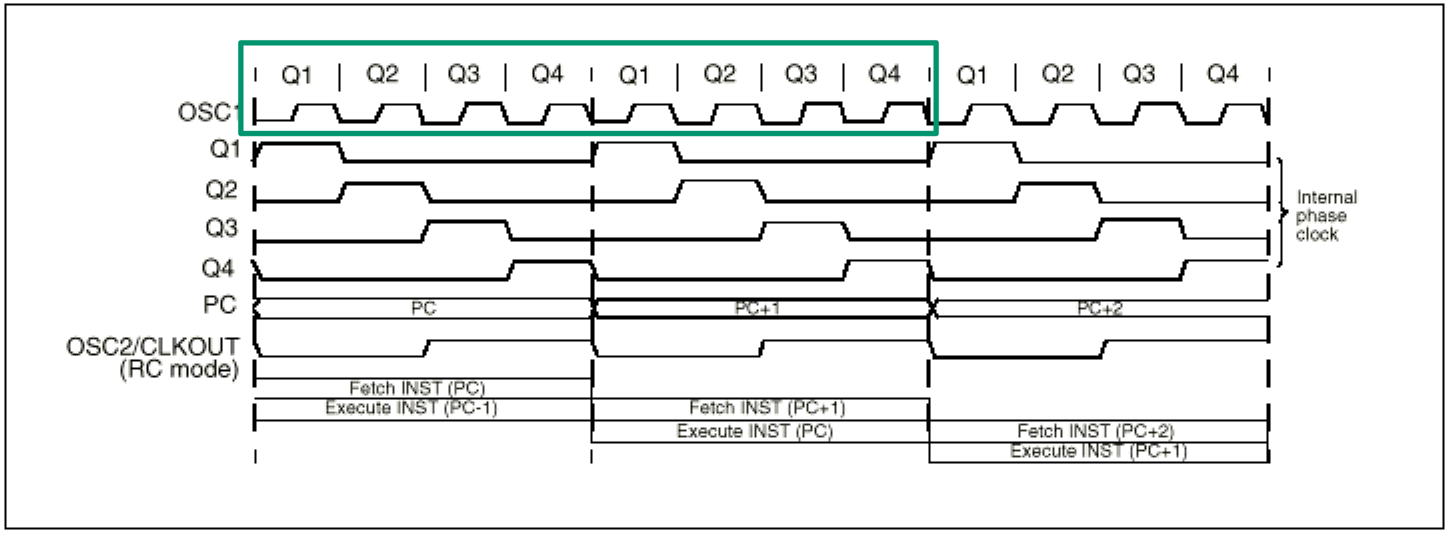
A car every minute is leaving the band – does it take one minute to build a car?

No at KIA's factory outside Zilina it will take 18 manhours to build a car (this is worldrecord! Toyota will need 30 manhours).

The solution is a **Pipeline**. 18 hours is 1080 minuts, så build is done in parallell at 1080 one minute stations. The factory has 3000 employees working in three shifts, ie 1000 workers per shift. Many of the station are thus completely robotized.

Fetch and Execute

FIGURE 3-3: CLOCK/INSTRUCTION CYCLE



PIC has Harvard architecture and can therby **Fetch** an instruktion *at the same time* it is **Executing** the previous instruction. It will take ***8 clock cykles*** to finnish an instruction. We have a **two step pipeline**, so there will be one instruction *finnished* after each fourth oscillator-clockcykel. With a 4 MHz clock this is 1.000.000 instructions/sec. Each instruction will take **1 µs**.

Instruction format

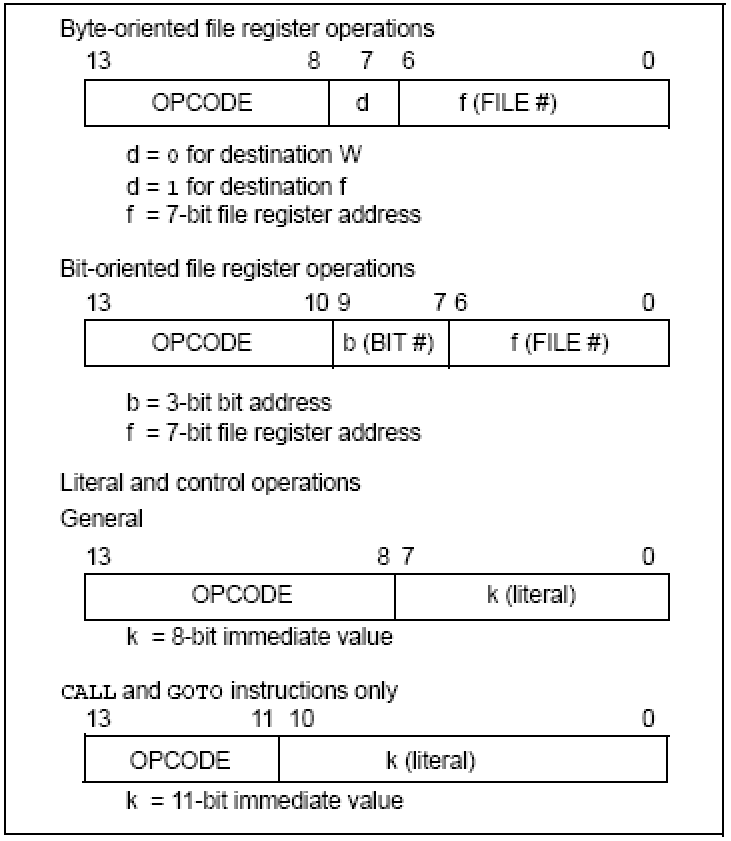
PIC is a classisc RISC-processor with only 33 instructions …

Instructions are **14** bit

• OP-code *what* to be done – is **6** bit (or 3 bit).

• The rest of the bits are used to tell – *with what* it should be done.

FIGURE 15-1: GENERAL FORMAT FOR INSTRUCTIONS


Byte-oriented file register operations
13
8
7
6
0
OPCODE
d
f (FILE #)
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address
Bit-oriented file register operations
13
10 9
7 6
0
OPCODE
b (BIT #)
f (FILE #)
b = 3-bit bit address
f = 7-bit file register address
Literal and control operations
General
13
8 7
0
OPCODE
k (literal)
k = 8-bit immediate value
CALL and GOTO instructions only
13
11 10
0
OPCODE
k (literal)
k = 11-bit immediate value

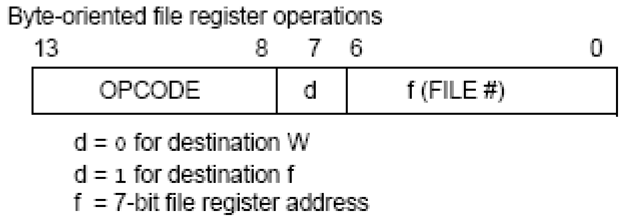
Byte operations

Ex. Addition of numbers in **FILE**, data memory, and working-register **W**. The result is stored lagras in workingregister or data memory – and the initial number will be overwritten.

```
ADDWF  f,d
ADDWF  f,0;  W=f+W
```

eller

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


Byte-oriented file register operations
13
8
7
6
0
OPCODE
d
f (FILE #)
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address

In the same way: `SUBWF f,d`

Assembler instructions are written as easy to remember abbreviation **mnemonics**.

More Byte operations

Some special cases of addition and subtraction, increase by one respective decrease by one, have their own instructions. Like the reset of register.

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

If you want to copy content between the memory and the working register one does it with

`MOVF f,0; W=f`

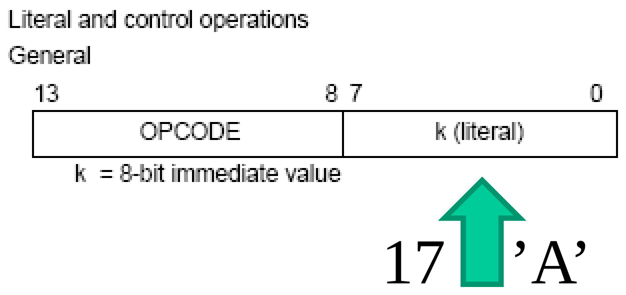
or between working register and memory with

`MOVWF f; f=W`

Move mean really Copy!

Program constants

Programconstants as number 17 or the letter ’A’ are stored inside instructions.

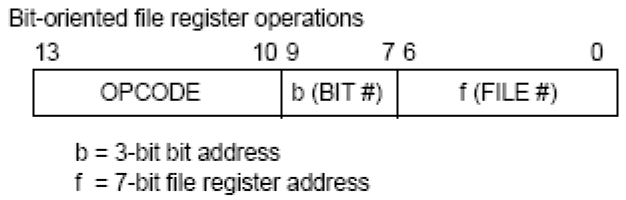


k is a ”**Literal**”, a Byte constant, stored inside the instruction **`MOVLW k; W=k.`** At the execution of the instruction the constant will be transfered to the working register.

More Literal-instructions: **`ADDLW k; W=W+k`**
`SUBLW k; W=W-k`

Bit operations

PIC processor has direct bit operations.

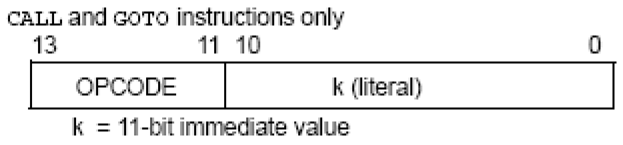


Bit-oriented file register operations
13
10 9
7 6
0
OPCODE
b (BIT #)
f (FILE #)
b = 3-bit bit address
f = 7-bit file register address

`BCF f,b` Clear bit **b** in File nr **f** (bits are numbered 0…7)
`BSF f,b` Set bit **b** in **f**

Program jumps

`GOTO k`	Program jump
`CALL k`	Subroutine call
`RETURN`	Return jump



Instruction **`GOTO`** changes PC to the value of Literal `k` which for this instruction is **11** bit (and two extra bits from register **`PCLATH`**). PC now continnues to exequte the program from the new place.

When **`CALL`** instruction, first the PC value is stored in a *stack register*, then its the same as with **`GOTO`**. At instruction **`RETURN`** the previous value of PC is retrieved from the stack register and the program continnues with the instruction that follows after the CALL instruction.

Conditional tests, skip

PIC processor has some instructions to test whether conditions are met and, if so, skip, the next instruction. The next instruction is then usually a GOTO instruction.

`DECFSZ f,d;` f - 1 but skip ”next” *if* 0-result
`INCFSZ f,d;` f +1 skip *if* 0 (registers can ”turn around”!)
`BTFSC f,b;` skip *if* bit b in f is 0 (Clear)
`BTFSS f,b;` skip *if* b in f is 1 (Set)

This counterintuitive thinking ”don’t jump if ..." is a bit special for PIC and no longer common to other processor types.

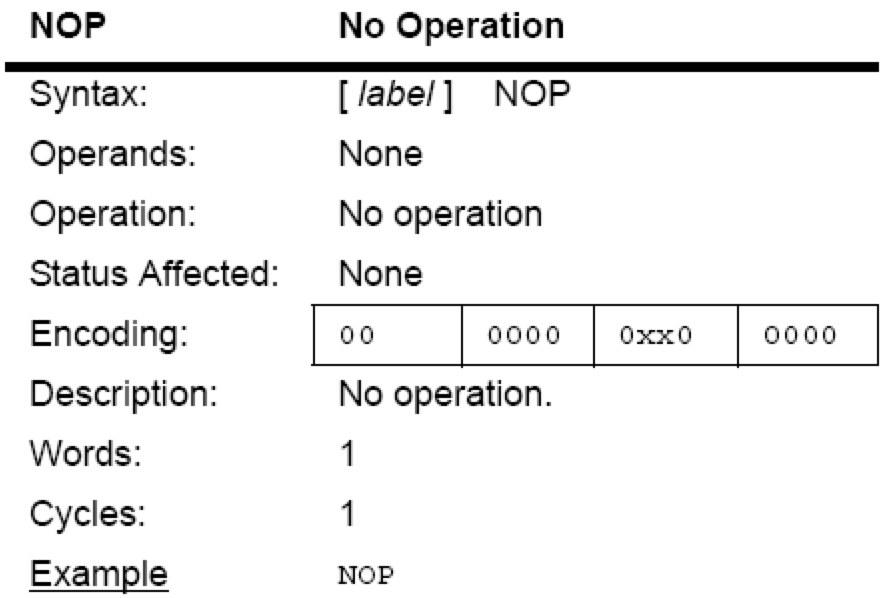
Why skip?

The outcome of a test often means that one needs to do an additional instruction that one would not otherwise do.

skip instruction skips this extra instruction, and because jumps always takes twice as long as other instructions, so take the instruction sequence always the same time to execute regardless of the result!

This can be seen as a feature of the PIC processor's instruction set.

NOP   No Operation



NOP	**No Operation**
Syntax:	[*label*] NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Encoding:	00 | 0000 | 0xx0 | 0000
Description:	No operation.
Words:	1
Cycles:	1
Example	NOP

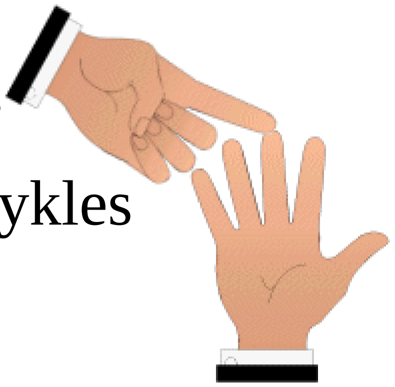
Processors generally have an instruction that does "nothing". It can be added to equalize the time differences between different paths in the program.

How long time does instructions take?

The processor internal clock uses ¼ of the oscillator frequency. Usual is 4 MHz crystal and then there will be 1 MHz clock speed. Most operations are performed in one clock cycle, ie, takes **1μs**. The instructions that affect the PC takes two clock cycles, ie, **2 μs**.

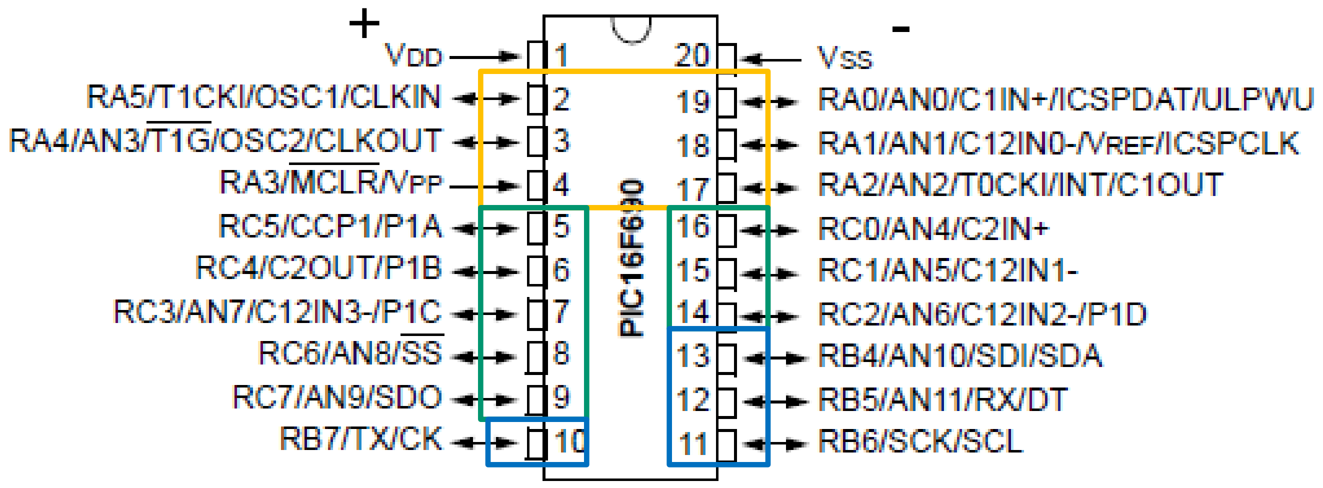
`GOTO, CALL, RETURN` Allways take 2 cycles,

`DECFSZ, INCFZ, BTFSC, BTFSS` takes 2 cykles when they create ”skip”, otherwise 1 cykle.



One can calculate the PIC processor execution time with finger counting!

Ports



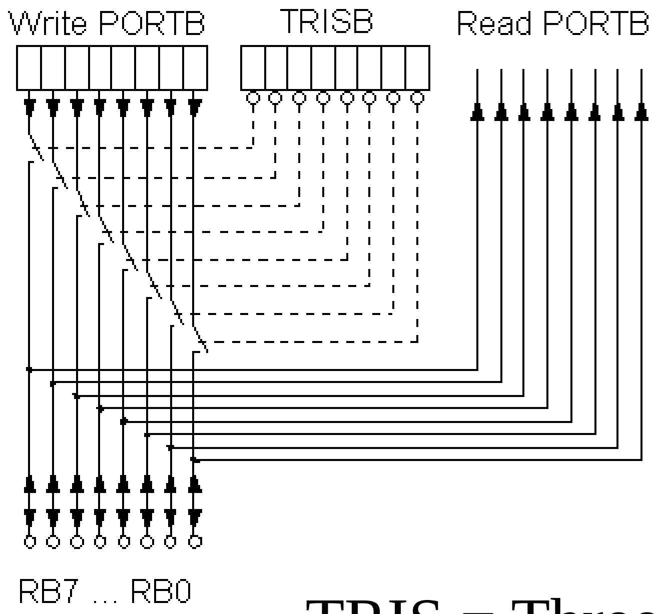
Of the PIC circuit pins 6 are bundled to a **PORTA** and 8 to a **PORTC**, 4 to a **PORTB**. The pins can also be used alone, and apparently they can have many optional features.

Tris-register

If a pin is to be used as **input** or **output** depends on settings in a TRIS-register.

`TRISA` and **`TRISB`** and **`TRISC`**

If the "corresponding" bit in trisregistret is **1** the pin is used as an input, if it's **0** as an output!



TRIS = Threestate

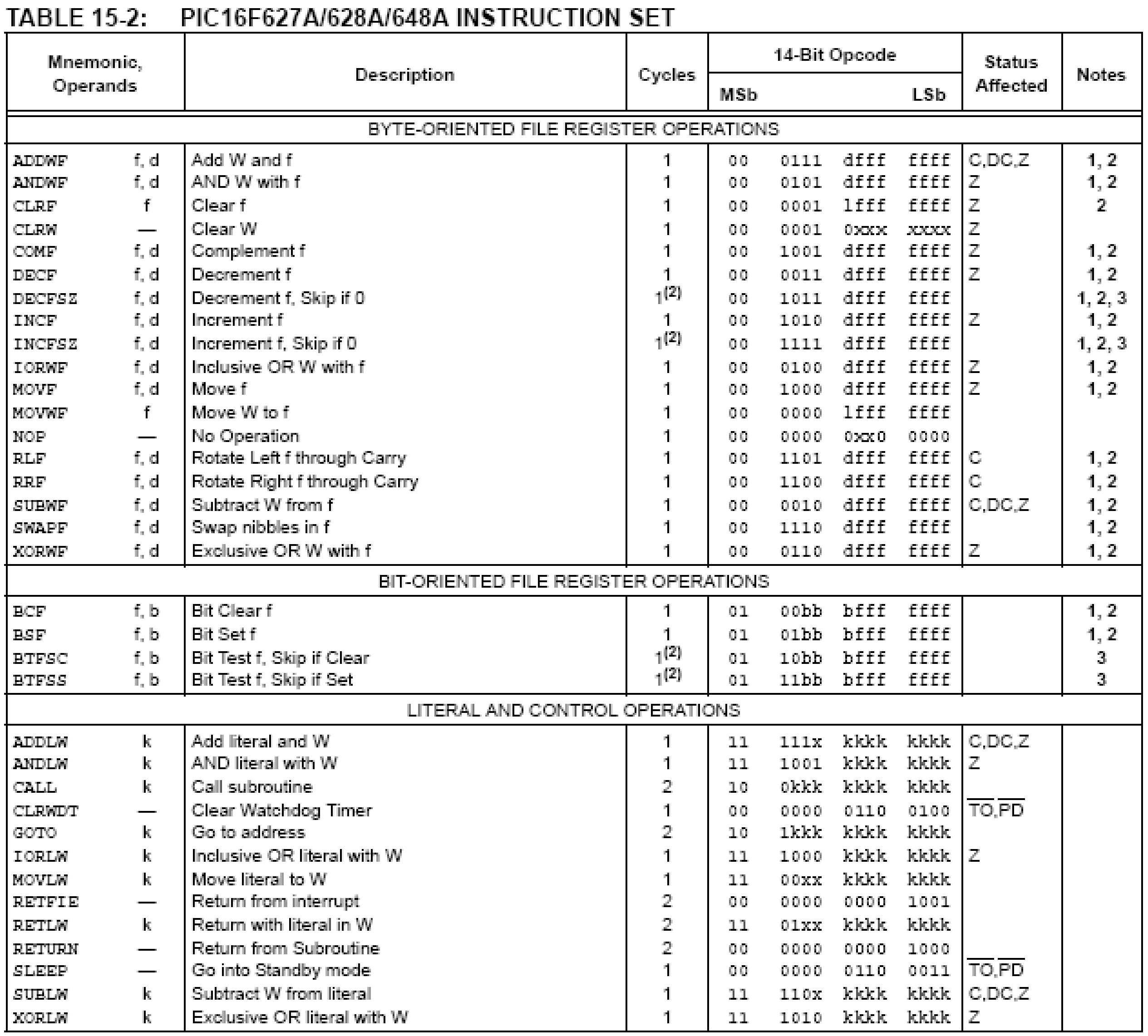
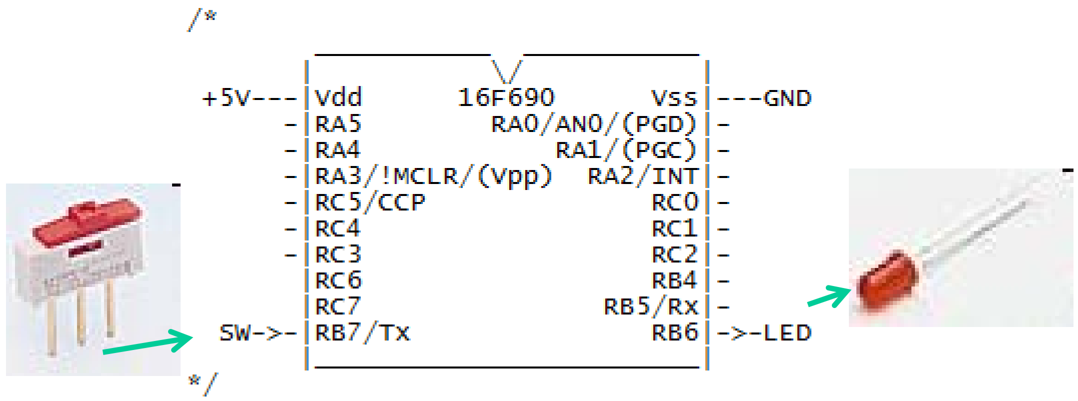


TABLE 15-2: PIC16F627A/628A/648A INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
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	lfff	ffff	Z	2
CLRW	—	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	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	lfff	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	$\overline{TO}$,$\overline{PD}$	
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	$\overline{TO}$,$\overline{PD}$	
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	

An Assembly program

```
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;
```

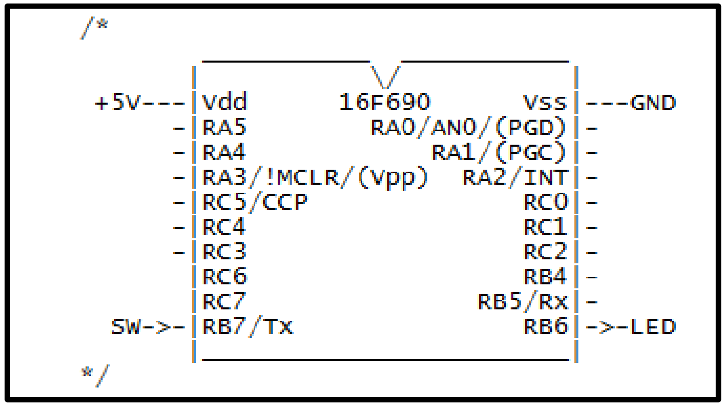


```
/*
         _____________  _____________
        |             \/             |
 +5V---|Vdd        16F690         Vss|---GND
      -|RA5            RA0/AN0/(PGD)|-
      -|RA4                RA1/(PGC)|-
      -|RA3/!MCLR/(Vpp)    RA2/INT  |-
      -|RC5/CCP                  RC0|-
      -|RC4                      RC1|-
      -|RC3                      RC2|-
       |RC6                      RB4|-
       |RC7                   RB5/Rx|-
 SW->-|RB7/Tx                    RB6|->-LED
       |____________________________|
*/
```

The program lights on and off the LED on the command from the switch.

(This of course could be done without PIC - but then it's no sport!)

Commented assembly program

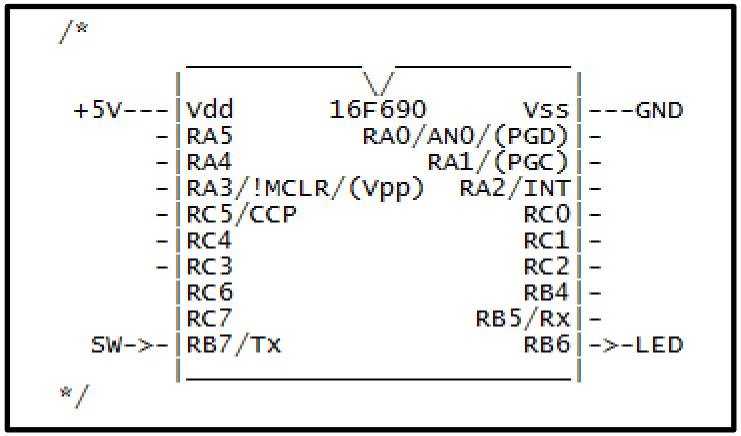


```
/*
              _____________  _____________
             |             \/             |
     +5V---|Vdd        16F690          Vss|---GND
          -|RA5             RA0/AN0/(PGD)|-
          -|RA4                 RA1/(PGC)|-
          -|RA3/!MCLR/(Vpp)     RA2/INT|-
          -|RC5/CCP                   RC0|-
          -|RC4                       RC1|-
          -|RC3                       RC2|-
           |RC6                       RB4|-
           |RC7                    RB5/Rx|-
     SW->-|RB7/Tx                     RB6|->-LED
             |___________________________|
*/
```

Assembly language program is called "spaghetti programming". It becomes easier to follow the program jumps when you draw out the arrows.

```
  init
    CLRF PORTB;       reset register portB
    MOVLW 10111111b;  get a constant to the working register W
    MOVWF TRISB;      copy the constant to trisB register
  loop
    BTFSS PORTB,7;    skip next instruction if portb.7 = 1
    GOTO lampoff;     jump to ”lampoff”
  lampon
    BSF PORTB,6;      Set portB.6 -> light up LED
    GOTO loop;        go on from ”loop”
  lampoff
    BCF PORTB,6;      reset portB.6 -> turn off LED
    GOTO loop;        go on from ”loop”
    end;
```

C-program



```
/*
           _____________  _____________
          |             \/             |
  +5V---|Vdd        16F690          Vss|---GND
       -|RA5          RA0/AN0/(PGD)|-
       -|RA4              RA1/(PGC)|-
       -|RA3/!MCLR/(Vpp)  RA2/INT|-
       -|RC5/CCP                RC0|-
       -|RC4                    RC1|-
       -|RC3                    RC2|-
        |RC6                    RB4|-
        |RC7                 RB5/Rx|-
   SW->-|RB7/Tx                 RB6|->-LED
        |__________________________|
*/
```

```
/* 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 – extensions of theC-language
Bitvariables **`variabel.bit`**
The compiler recognizes names of most registers, the rest of the names are stated in the processor include file.

Download format

The program code is downloaded to the chip with a circuit programmer.

The format used is a text file with the op-codes as a string of Hex digits. This is the download code for the previous example program.

```
:10000000128316031307108312071483120313A6
:10001000871C0C2807140628831203130710062 8D0
:02400E00D400DC
:00000001FF
```

End of file.

Compilation ”report”

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 internal variables)

Built-in the compiler provides the following names of registers and flags (bits in 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;
```

These should not be declared in the programs. Include files then hold additional register names and names of bits, the same names that are used in the official manual.

(Cc5x internal functions)

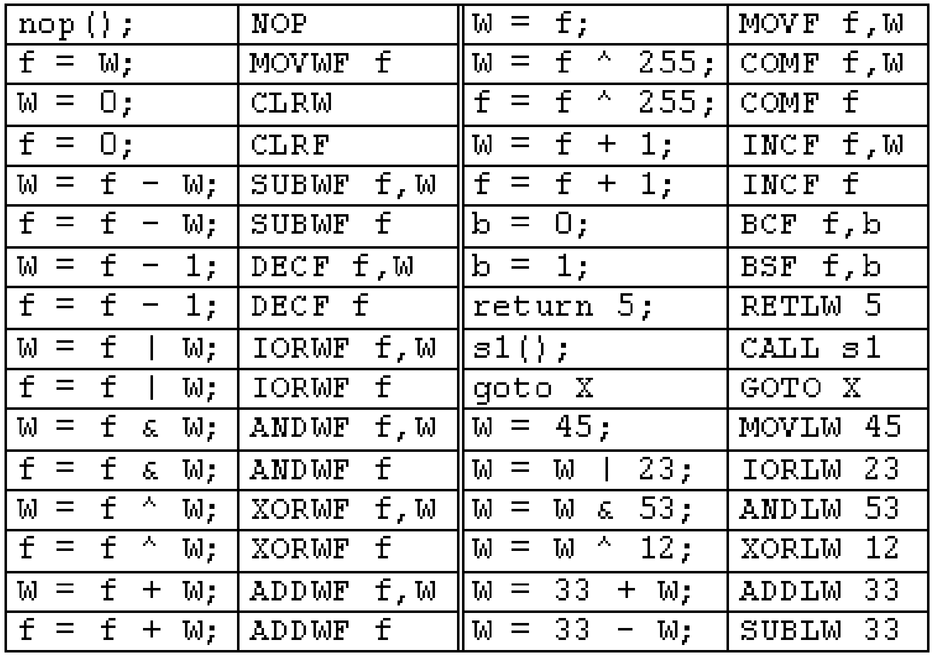
The internal functions provide "direct access" to some of the PIC processor instructions:

```
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);` An internal function that can be called to reset all data memory in the processor.

(Simple C-statements → Assembler)

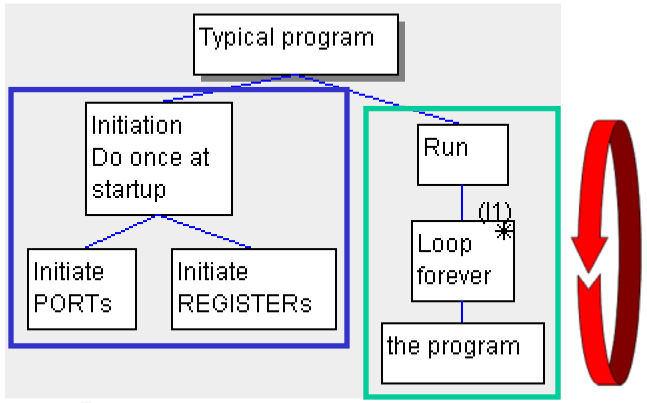
Simple C statements are in general translated directly to the single assembler instructions. Programs written in assembly language can be translated instructions by instruction to a Cc5x C program.

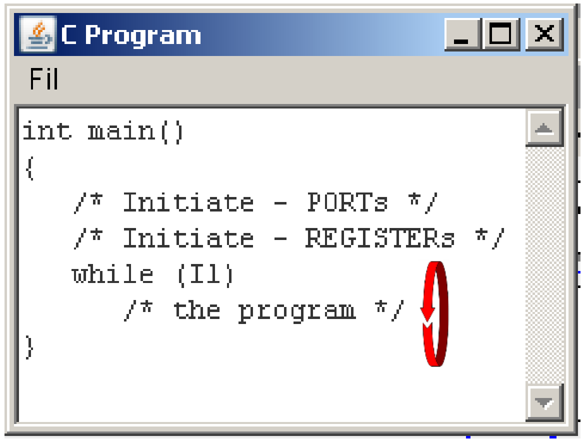


`nop();`	`NOP`	`W = f;`	`MOVF f,W`
`f = W;`	`MOVWF f`	`W = f ^ 255;`	`COMF f,W`
`W = 0;`	`CLRW`	`f = f ^ 255;`	`COMF f`
`f = 0;`	`CLRF`	`W = f + 1;`	`INCF f,W`
`W = f - W;`	`SUBWF f,W`	`f = f + 1;`	`INCF f`
`f = f - W;`	`SUBWF f`	`b = 0;`	`BCF f,b`
`W = f - 1;`	`DECF f,W`	`b = 1;`	`BSF f,b`
`f = f -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

Typical program structures

A typical program





```
int main()
{
   /* Initiate - PORTs */
   /* Initiate - REGISTERs */
   while (I1)
      /* the program */
}
```

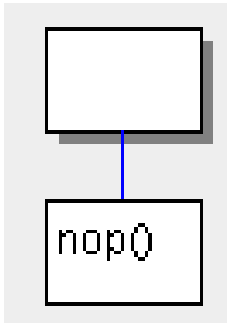
A typical program.

First initiate PORTs and units so they are set to fit the application. This is done **once for all** in the beginning of the program.

Then gthe program loops for ever – and reacts on input signals and delivers output signals for every turn in the loop.

The program finnishes when the power is turned off.

Single run program?



- C-program:

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

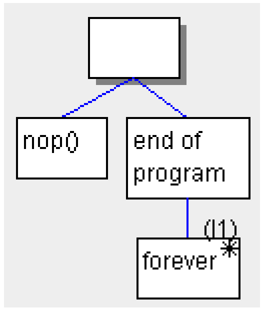
- Translated to assembly:

```
main
                                ;  nop(); /* to do something once */
        NOP
                                ;}
        SLEEP
        GOTO main

        END
```

Single run program would not work, the compiler inserts **SLEEP** command, so the processor enters current save mode.

This also goes for the IO-units.

Single run program?



- C-program:

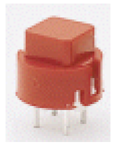
```
void main( void)
{
  nop(); /* something once */
  while(1);
}
```

- Translated to assembly:

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

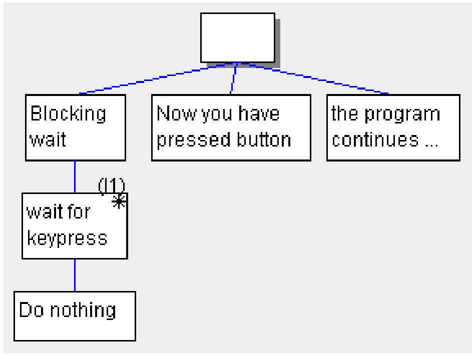
This is a program that does not force the compiler to use **SLEEP**, the power saving mode.

Wait for a key press?



PORTB bit 0 gets 1 when you press

Many times the CPU has not so much to do, then you can use blocking code.



- wait for a key press, blocking code:

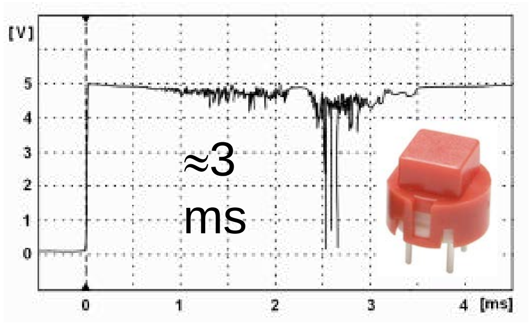
```
while (PORTB & 0x01 == 0) /* do nothing */ ;
/* OK, now you have pressed the button ... */
```

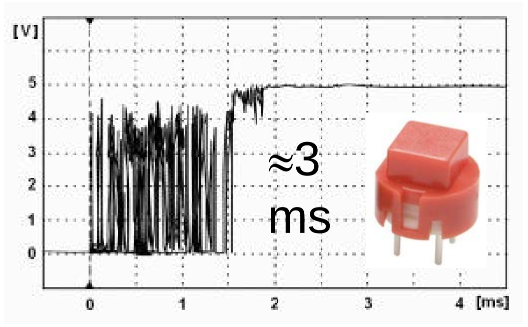
- Or simpler – PIC-processors have bitvariables:

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

/* OK, now you have pressed the button ... */
```

Contact bounces!

When you press, or release, a mechanical contact it bounces a while before the contact surface is coming to rest. PIC processor are so fast that they can perceive each bounce as distinct contact press!



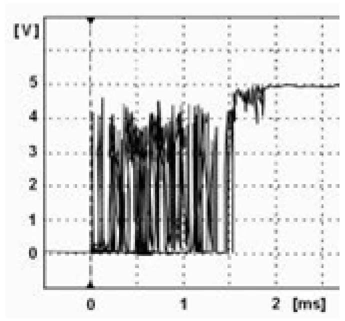


If a contact will bounce much or little is not visible on the outside!

Toggle a LED ON/OFF

Nothing else than a random number generator, anything can happen/not happen when you press the button!

```
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 */
   }
}
```



- **Not as thought, every other time - but a random number generator!**

Toggle a LED ON/OFF

Wait out the contact bounces. A contact can bounce both when pressing it and when you release it!

```
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);
     while( PORTB.7 ) ;   /* wait for key released */
     delay(5);
   }
}
```

Wait out the contact bounces (>5ms)

Wait out the contact bounces (>5ms)

- **Now it works!**

`delay()` function

C-functions

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

- Function declaration (prototype) before 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);
     while( PORTB.7 ) ;   /* wait for key released */
     delay(5);
   }
}
```

- Function call
- Function call

- Place the funktion definition after main() in the same file.

`delay()` function

- Place function definitions after main() in the same file.

```
/* 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
Nr of turns

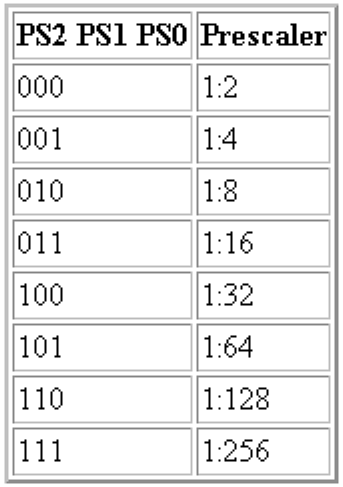
1000 μs

It is the after-tested loop that is the iteration procedure that best fits the PIC processor.

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

TIMER0

TIMER0 is an internal 8-bit modulo 256-counter which can be read/written from program. When the timer “turns around” the bit **T0IF** is set.
If bit **TOCS** in **OPTION** register is "0" then the processor clock is counted. If bit **TOCS** is "1" then edges on pin **T0CKI** is counted.

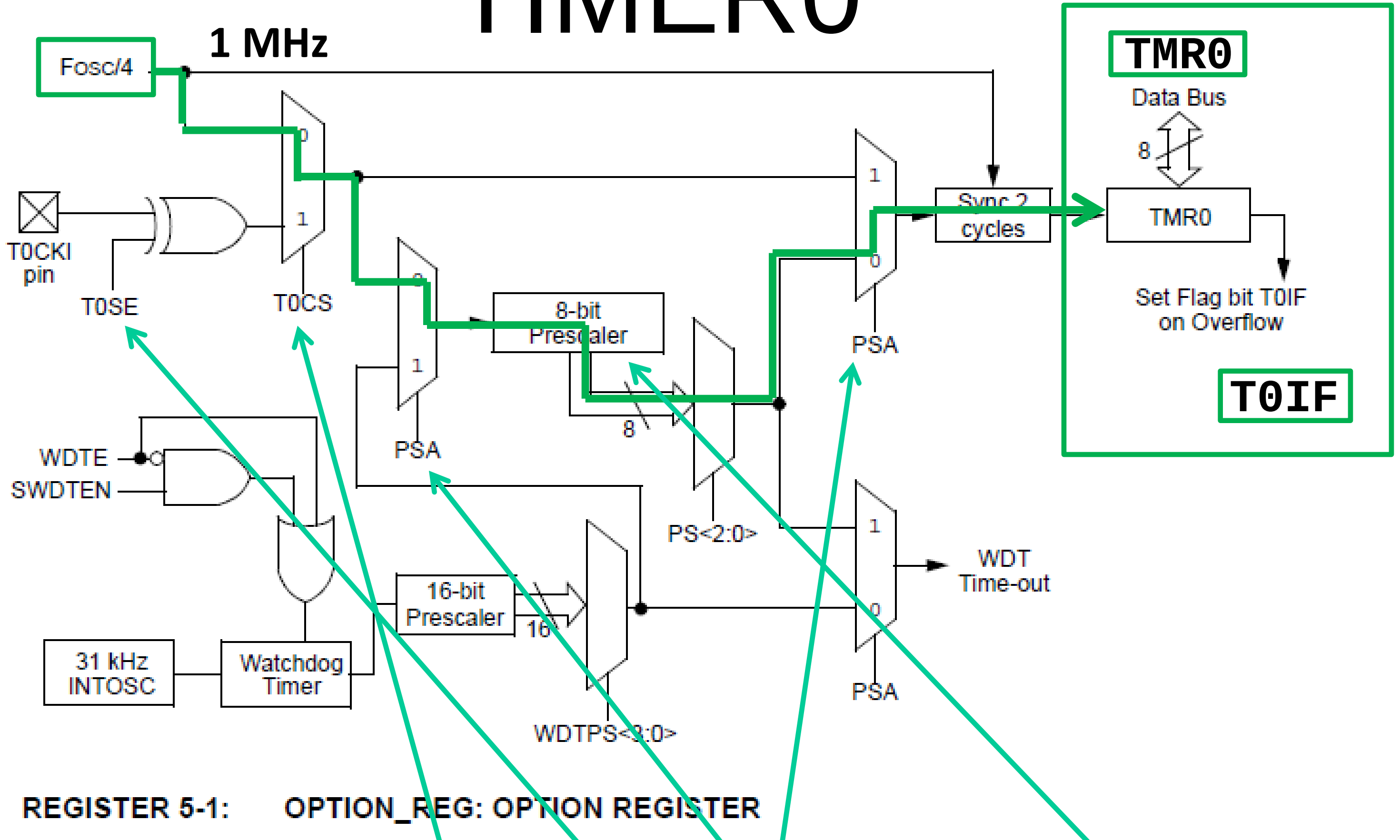


PS2 PS1 PS0	Prescaler
000	1:2
001	1:4
010	1:8
011	1:16
100	1:32
101	1:64
110	1:128
111	1:256

The bit **PSA=0** inserts a **prescaler**, a frequency divider. With it active only a fraction of the incoming pulses are counted. Bits **PS2 PS1 PS0** sets the prescaler division ratio.

```
TMR0=0;    /* reset timer0 */
time=TMR0; /* store timer0 value in char variable time */
TMR0=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
$\overline{\text{RABPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

C-functions summary

- Function deklarations before **`main()`**.
- Call from inside **`main()`**or from inside other functions.
- Function definitions afterr **`main()`**, in the same file.

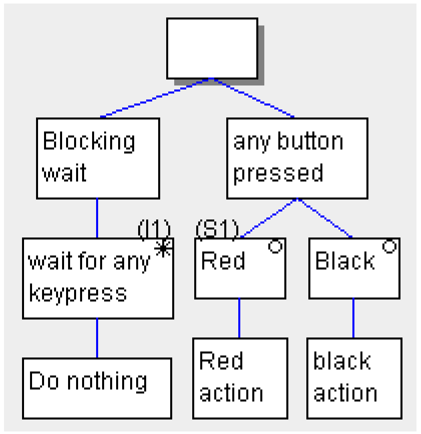
Often its so little code that everything can be in one file. The functions are often tailored directly to the application and the processor, therefore it may be unnecessary to store them as a ”general” function library.

Wait for key presses?

PORTB bit 0
gets 1 when one
presses the key

PORTB bit 1
gets 1 when one
presses the key

Two keys, **blocking code.**



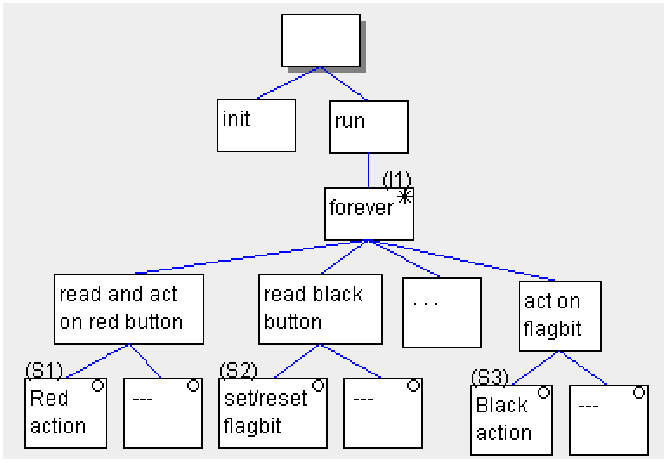
OR

```
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 */ ;
```

React on keypresses?

PORTB bit 0
gets 1 when one
presses the key

PORTB bit 1
gets 1 when one
presses the key



Two keys, **nonblocking code**

```
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       */ ;
 }
```

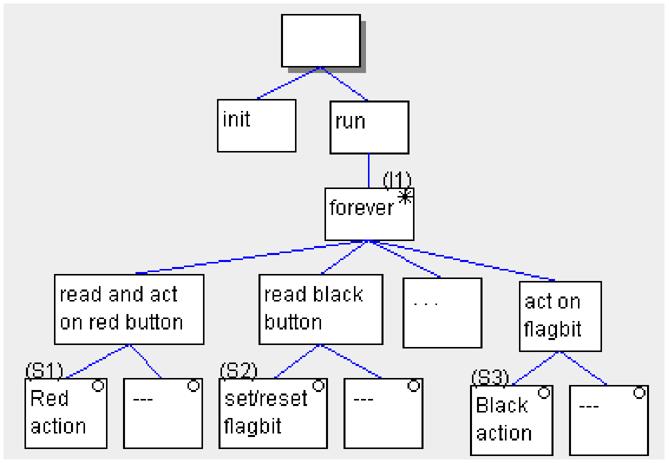
• **Contact bounces?**

One can react directly on the key status or share the information with a bitvariabel, a flag bit.

React on keypresses?

PORTB bit 0
gets 1 when one
presses the key

PORTB bit 1
gets 1 when one
presses the key



Two keys, **nonblocking code**

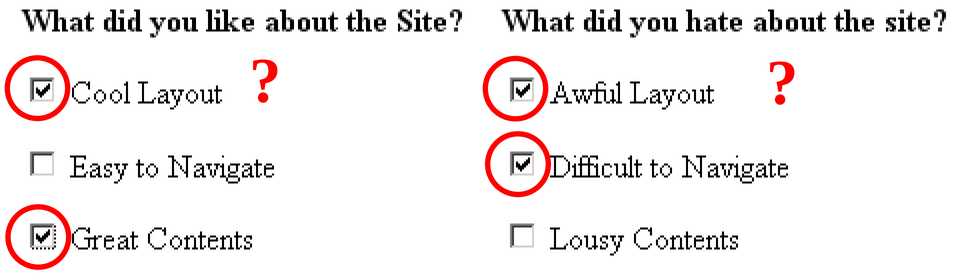
```
bit flagbit;
While(1) /* main programloop */
 {
   /* examine button status */
   if(PORTB.0) /* direct action for red button  */ ;
   if(PORTB.1) flagbit = 1; else flagbit = 0;
   /* . . .  */
   /* later, act on the flagbit                  */
   if(flagbit) /* action for black button       */ ;

   delay(5);
 }
```

Wait out (>5ms) contact bounces
before the nect turn in the main-loop

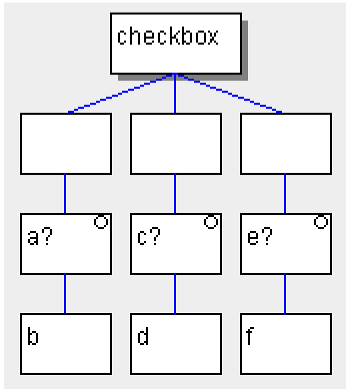
Checkbox or Radiobutton?

Checkbox (meny alternatives):

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

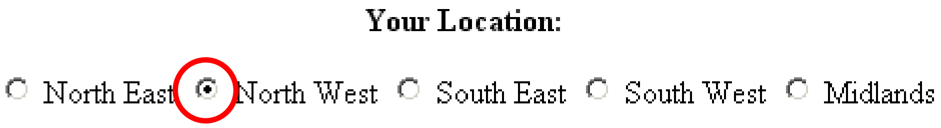


What did you like about the Site?	What did you hate about the site?
[✓] Cool Layout ?	[✓] Awful Layout ?
☐ Easy to Navigate	[✓] Difficult to Navigate
[✓] Great Contents	☐ Lousy Contents



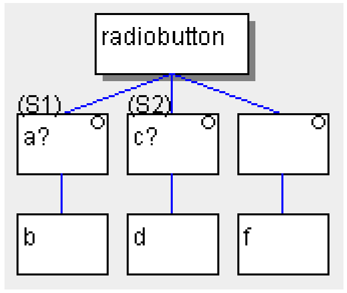
Radio Button (only one):

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



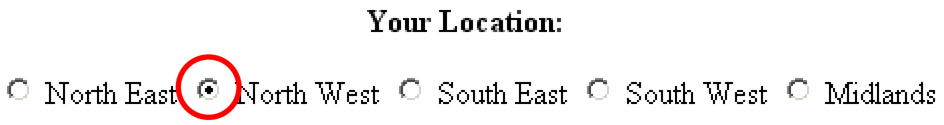
Your Location:

○ North East [✓ ● North West] ○ South East ○ South West ○ Midlands



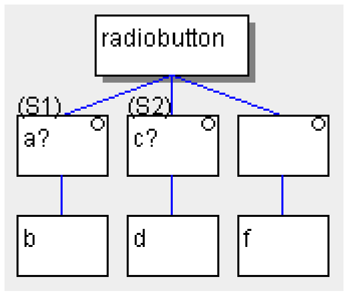
Radiobutton …

To select only one option among several …


Your Location:

○ North East [✓ ● North West] ○ South East ○ South West ○ Midlands

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



Or with the C-language **switch-case** expression …

C-language **switch – case** expression

Hint! Note that *B Knudsen* compiler generates more effective code for

- **`switch()`** – **`case`**

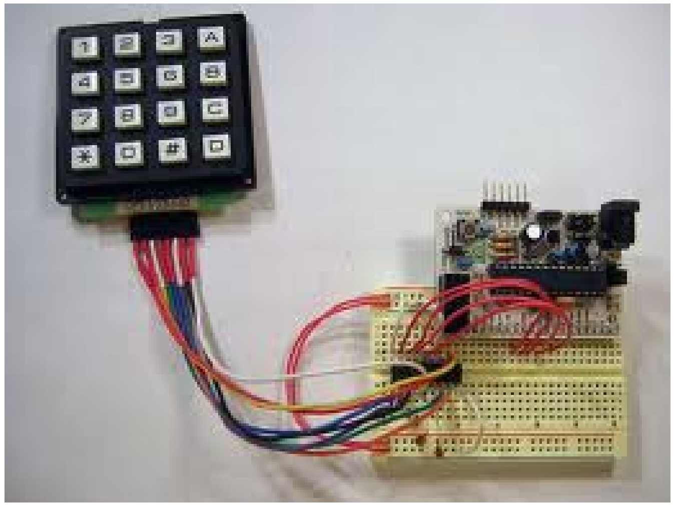
than for

- **`if()`** – **`else if()`** – **`else`**

so always use a switch statement when possible!

C 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=' ';
}
```



Recoding. Keyboard delivers mostly a completely different code **d** than is engraved on the key **k** !

Handy menu-handling

```
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");
    }
```

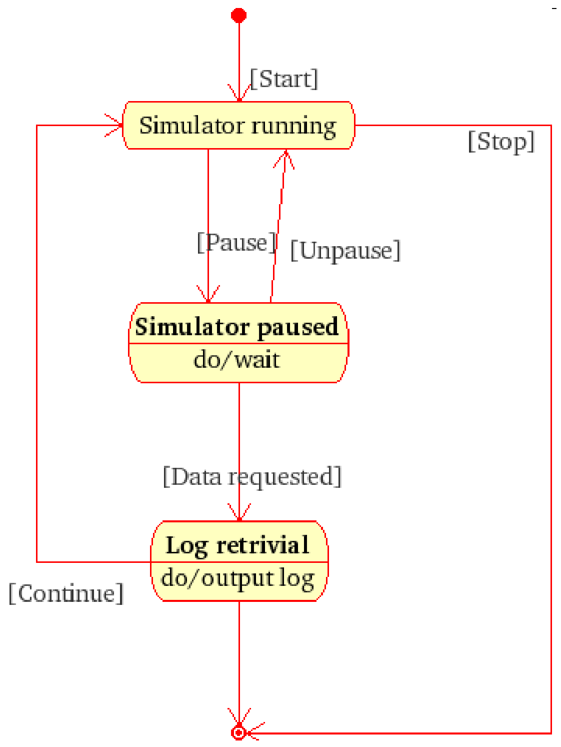
Group alternatives

Default, for all unspecified alternatives

Programing with state chart

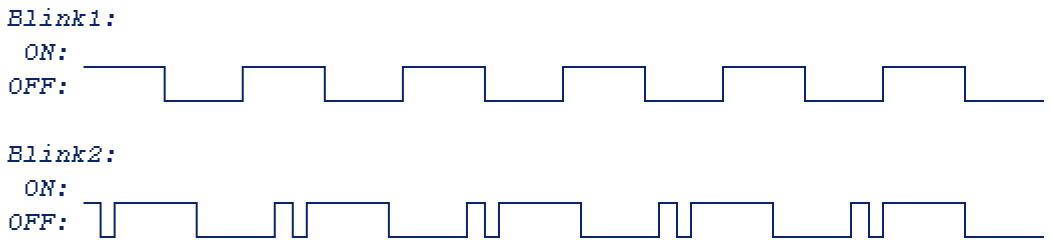
A very common technique for programming embedded processors is to use "state" and "state chart".

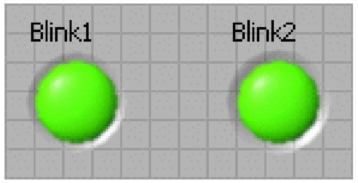
The idea is borrowed from Digital Designs ”state machines".



UML-state chart

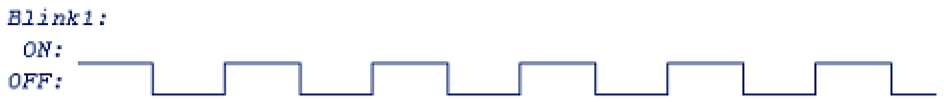
Multitask?

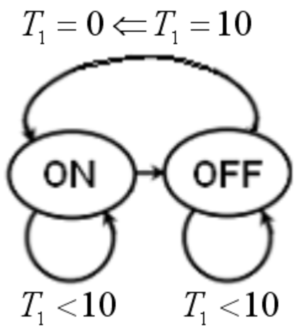




```
/* Blink1:  1s ON - 1s OFF  */
/* Blink2:  0,2s ON - 0,2s OFF - 1s ON - 1s OFF  */
```

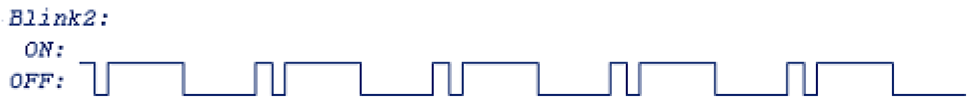
First one lightdiode …

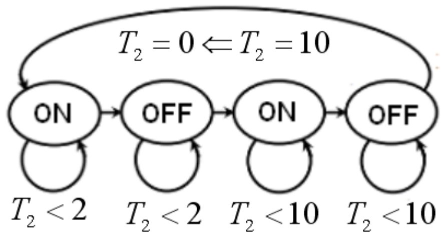




```
while(1)
 {
   /* Blink1:  1s ON - 1s OFF  */
   switch(State1)
    {
      case 0:
        PORTB_copy.6=1;    /* Blink1 = ON */
        Time1++;
        if( Time1 == 10 ) { State1 = 1; Time1 = 0; }
        break;
      case 1:
        PORTB_copy.6=0;    /* Blink1 = OFF */
        Time1++;
     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

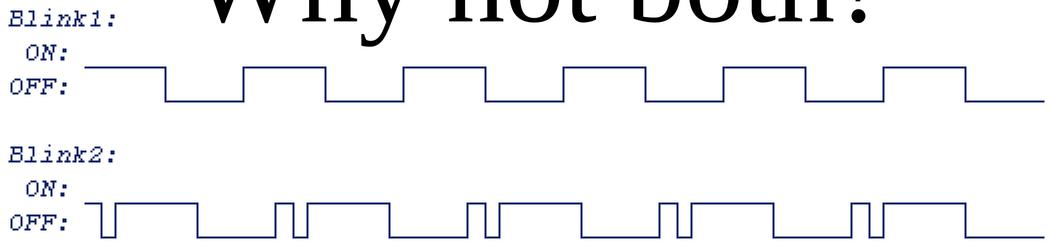
Then another lightdiode …



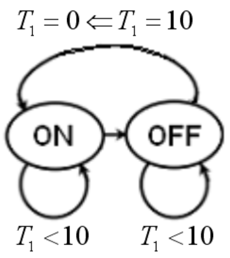


```
while(1)
 {
   /* Blink2:  0,2s ON - 0,2s OFF - 1s ON - 1s OFF */
   switch(State2){
      case 0:
        PORTB_copy.5 = 1; Time2++;  /* Blink2  ON */
        if( Time2 == 2 ) { State2 = 1; Time2 = 0; }
        break;
      case 1:
        PORTB_copy.5 = 0; Time2++;  /* Blink2  OFF */
        if(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

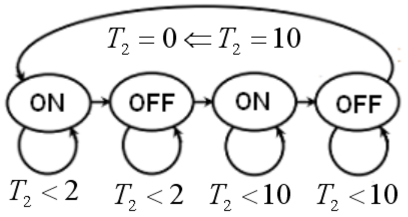
Why not both?



```
while(1)
 {
  /* Blink1:  1s ON - 1s OFF  */
  switch(State1)
   {
     case 0: ... ; break;
     case 1: ... ;
   }
  /* Blink2:  0,2s ON - 0,2s OFF - 1s ON - 1s OFF */
  switch(State2)
   {
     case 0:  ... ; break;
     case 1:  ... ; break;
     case 2:  ... ; break;
     case 3:  ... ;
   }
  PORTB = PORTB_copy;
  delay10(10); /* 0,1 sek delay */
 }
```



fast 10 µs

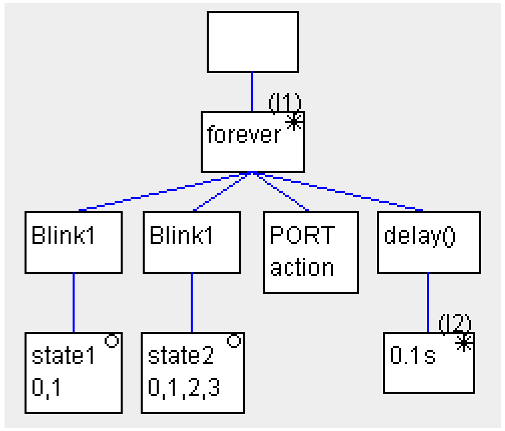


fast 10 µs

slow 0.1 s

State machine

By programming "state machines" (compare with Digital Design) you can make it look as if the processor is able to accomplish many things simultaneously. One can try out each thing separately, and usually works then the whole combination as intended.



WARNING! There is a "sneaky" so-called **RMW problem**.
HINT, SOLUTION: Changing bits in a variable **`PORT_copy`** instead of directly on the **`PORT`**. Then copy this entire variable to port, **`port = PORT_copy;`**
More about this later in course …