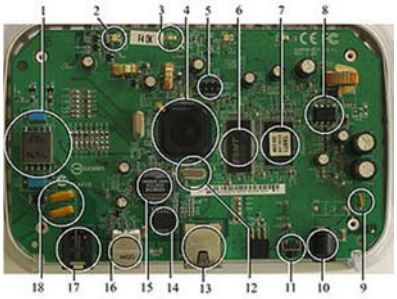
Embedded Systems

AVR Assembly Language Programming: I) assembler directives



Inside an ASDL Modem/Router

AVR, ATmega88PA, ATtiny45, etc.

- **We will cover the main AVR assembler directives and features**
  - → **complete Atmel documentation can be found under "Resources" on class website**
- **The AVR assembler supports different number notations**
  - → **quick review of number notations**

Review Number Notation

The AVR documentation and assembler use several notations for numbers, and we will follow these in the lecture notes.

By default, numbers are decimal. Thus "**10**" means "ten, decimal".

Hexadecimal numbers are written as **0xAB**, or **0xaB,** or **0xAb**. The leading character is a zero. Sometimes hexadecimal numbers are written as **$AB**.

Binary numbers are written as **0b10101011.** The leading character is a zero.

Octal numbers begin with "**0**" (zero). For example: **0253**

Integer Number Notation (8-bit)

BIN	HEX	Unsigned INT	Signed INT
0b00000000	0x00	0	0
0b00000001	0x01	1	1
…	…	…	…
0b01111111	0x7F	127	127
0b10000000	0x80	128	-128
0b10000001	0x81	129	-127
0b10000010	0x82	130	-126
…	…	…	…
0b11111110	0xFE	254	-2
0b11111111	0xFF	255	-1

N,Z,C Flags in SREG!

Signed INT: MSb → sign

```
42  = 0b00101010 = 0x2A
-42 = 0b11010110 = 0xD6
0b100000000
```

2's complement

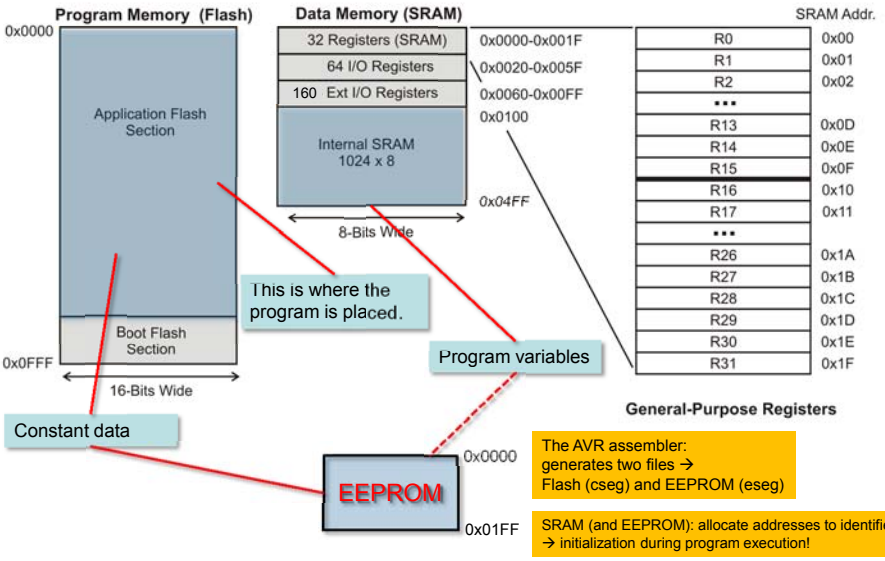
AVR Assembler

- **Assembler: tool to facilitate the generation of a sequence of bytes representing machine instructions (opcode) and data**
  - **Example:**

    .db 0x03, 0x95
    .dw 38147
    inc R16
- **The programmer should write statements that clearly communicate the intent**
  - **E.g., → Instructions vs. data**
- **The programmer needs to decide where the bytes will be placed in memory**
  - **Type of memory (Flash, SRAM, or EEPROM)**
  - **Memory address**
  - **→ see AVR memory map**

→ Assembler keeps track of where the next byte should be located

ATmega88 Memory Map



The AVR Assembler

- We will be using AVRASM2 assembler that comes with AVR studio
- The assembler performs the conversion from English MNEMONICS to OPCODES that are programmed into the microcontroller
  - For example:
    inc R16 → 0x9503
- "Assembler" ≠ "Compiler"
- C compilers generate assembly language mnemonics that are then assembled by an assembler

Example – Generate the Corresponding OPCODE & Store it in AVR Program Memory

Include definitions for Attiny45 (register names, …)

```
.include "tn45def.inc"
.cseg
.org 0

    ldi R16, 0xFA
    nop
    inc R16
```

put what follows in Flash memory

start at address 0x0000

Example – Generate the Corresponding OPCODE & Store it in AVR Program Memory

```
.include "tn45def.inc"
.cseg
.org 0

    ldi R16, 0xFA
    nop
    inc R16
```

LDI – Load Immediate

Description:

Loads an 8 bit constant directly to register 16 to 31.

	Operation:	Operands:	Program Counter:
(i)	Rd ← K		
(i)	Syntax: LDI Rd,K	16 ≤ d ≤ 31, 0 ≤ K ≤ 255	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK
0x E	F	0	A

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
clr   r31      ; Clear Z high byte
ldi   r30,$F0  ; Set Z low byte to $F0
lpm            ; Load constant from Program
               ; memory pointed to by Z
```

Words: 1 (2 bytes)
Cycles: 1

Example – Generate the Corresponding OPCODE & Store it in AVR Program Memory

```
.include "tn45def.inc"
.cseg
.org 0

    ldi R16, 0xFA
    nop
    inc R16
```

NOP – No Operation

Description:

This instruction performs a single cycle No Operation.

	Operation:	Operands:	Program Counter:
(i)	No		
(i)	Syntax: NOP	None	PC ← PC + 1

16-bit Opcode:

0000	0000	0000	0000
0x 0	0	0	0

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Example:

```
clr   r16      ; Clear r16
ser   r17      ; Set r17
out   $18,r16  ; Write zeros to Port B
nop            ; Wait (do nothing)
out   $18,r17  ; Write ones to Port B
```

Words: 1 (2 bytes)
Cycles: 1

Example – Generate the Corresponding OPCODE & Store it in AVR Program Memory

```
.include "tn45def.inc"
.cseg
.org 0

    ldi R16, 0xFA
    nop
    inc R16
```

INC – Increment

Description:

Adds one -1- to the contents of register Rd and places the result in the destination register Rd.

The C Flag in SREG is not affected by the operation, thus allowing the INC instruction to be used on a tiple-precision computations.

When operating on unsigned numbers, only BREQ and BRNE branches can be expected to perform operating on two's complement values, all signed branches are available.

Operation:

(i) Rd ← Rd + 1

	Syntax:	Operands:	Program Counter:
(i)	INC Rd	0 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode:

1001	010d	dddd	0011
0x 9	5	0	3

Status Register and Boolean Formula:

I	T	H	S	V	N	Z	C
-	-	-	⇔	⇔	⇔	⇔	-

S: N ⊕ V
For signed tests.

V: R7 •$\overline{R6}$ •$\overline{R5}$ •$\overline{R4}$ •$\overline{R3}$• $\overline{R2}$ •$\overline{R1}$ •$\overline{R0}$
Set if two's complement overflow resulted from the operation; cleared otherwise. Two's complen

Embe

Example – Generate the Corresponding OPCODE in the Program Memory of the AVR

```
.include "tn45def.inc"
.cseg
.org 0

    ldi R16, 0xFA
    nop
    inc R16
```

0xEF0A
0x0000
0x9503

AVR → LITTLE ENDIAN!

Memory 1

Memory: prog FLASH Address: 0x0000,prog

```
prog 0x0000  0a ef 00 00 03 95 ff ff ff ff ff ff ff ff ff ff  .ï....ÿÿÿÿÿÿÿÿÿÿ
prog 0x0013  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x0026  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x0039  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

The assembler's text substitution allows one to use symbols rather than numbers

```
.equ    PORTB   = 0x18
.equ    DDRB    = 0x17

.include "tn45def.inc"
```

```
.include "tn45def.inc"
.cseg

 sbi  DDRB,1

 loop:
   sbi   PORTB,1
   cbi   PORTB,1
   jmp   loop
```

Phase 1: text substitution

```
000000        sbi  0x17,1

000001        sbi   0x18,1
000002        cbi   0x18,1
000003        jmp   0x0001
```

These are program addresses

Phase 2: machine instructions,…

The assembler's text substitution allows one to use symbols rather than numbers

```
.include "tn45def.inc"
.cseg

 sbi  DDRB,1

 loop:
   sbi   PORTB,1
   cbi   PORTB,1
   jmp   loop
```

```
000000        sbi  0x17,1

000001        sbi   0x18,1
000002        cbi   0x18,1
000003        jmp   0x0001
```

```
.include "tn45def.inc"
.cseg

 sbi  DDRB,1
 sbi  DDRB,2

 loop:
   sbi   PORTB,1
   cbi   PORTB,1
   jmp   loop
```

```
000000        sbi  0x17,1
000001        sbi  0x17,2

000002        sbi   0x18,1
000004        cbi   0x18,1
000005        jmp   0x0002
```

The AVR Assembler

The AVR Assembler offers:

- Assembler Directives
- Assembler Functions
- Assembler Operators
- Assembler Preprocessor
- …

Assembler Directives

Assembler directives instruct the assembler.

The directives are not translated directly into opcode.

Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on.

```
.include "tn45def.inc"
.cseg

 sbi  DDRB,1

 loop:
   sbi   PORTB,1
   cbi   PORTB,1
   rjmp  loop
```

This does not generate opcode. Rather it directs the assembler to replace this line with the contents of "tn45def.inc"

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

We have seen this directive several times thus far

Note: *All directives must be preceded by a period.*

Assembler Directives

Directive	Description	Examples
.cseg **.dseg** **.eseg**	Switch to the specific memory segment: code (Flash), data (SRAM), or EEPROM • Can have several segments of the same type → will be concatenated • Each segment has their own location counter (16 bit) • The default segment is ".cseg"	**.cseg ; program code comes next** … **.dseg ; start data segment**

Assembler Directives

Directive	Description	Examples
.byte	Set aside address in **SRAM** or **EEPROM** where bytes can be stored during execution Parameter: # of bytes to reserve (Note: no initial values can be specified → initialization must be done by software during program execution)	**buff: .byte 32 ; a 32 byte area** **count: .byte 2 ; a word-sized counter**
.db **.dw**	Define a series of **byte** or **word** values that will be placed in **Flash** or **EEPROM** when downloaded to the µC	**lookup_table: .db 13, -5, 42, 0x80, 0xFF** **maxinit: .dw 32767** **str: .db "hello" "there"**

Note the use of the address labels → needed to be able to refer to the corresponding memory location!

Example

Note: A Data Segment will normally only consist of BYTE directives (and labels).

Example:

```
.DSEG
        var1:   .BYTE 1         ; reserve 1 byte to var1
        table:  .BYTE tab_size  ; reserve tab_size bytes
.CSEG
        ldi     r30,low(var1)   ; Load Z register low
        ldi     r31,high(var1)  ; Load Z register high
        ld      r1,Z            ; Load VAR1 into register 1
```

Example

Important: initialize in program before use!

Example:

```
.DSEG                           ; Start data segment
        vartab:   .BYTE 4       ; Reserve 4 bytes in SRAM
.ESEG
        eevar:    .DW 0xff0f    ; Initialize one word in
                                ; EEPROM
.CSEG                           ; Start code segment
        const:    .DW 2         ; Write 0x0002 in prog.mem.
        mov r1,r0               ; Do something
```

Important: program flash memory (16bit) and EEPROM (8bit)!

Example

Example:

```
.CSEG
        consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
        eeconst:.DB 0xff
```

.DB → Each expression must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8 bits two's complement of the number will be placed in the program memory (flash) or EEPROM memory location.

Example

Example:

```
.CSEG
        varlist:.DW 0,0xffff,0b1001110001010101,-32768,65535
.ESEG
        eevar:  .DW 0xffff
```

.DW → The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -32768 and 65535. If the expression evaluates to a negative number, the 16 bits two's complement of the number will be placed in the program memory location.

Assembler Directives

Directive	Description	Examples
.def	Define a symbolic name for a register. Note: • A register can have several symbolic names attached to it • A symbol can be redefined later in the program	**.def sum=R16** **.def counter=R24**
.undef	Un-define a register name	**.undef sum**
.equ	Set a symbol equal to a value (or expression); **→symbols are constant!**	**.equ tabsize=14** **.equ cr=0x0D**
.set	Set a symbol to a value (or expression); **→ set symbols can be reassigned!**	**.set flag=1** **.set flag=flag+1**

Example - .def

Example:

```
.DEF temp=R16
.DEF ior=R0
.CSEG
        ldi     temp,0xf0     ; Load 0xf0 into temp register
        in      ior,0x3f      ; Read SREG into ior register
        eor     temp,ior      ; Exclusive or temp and ior
```


Example - .equ

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
.CSEG                           ; Start code segment
        clr     r2              ; Clear register 2
        out     porta,r2        ; Write to Port A
```

Example - .set

Example:

```
.SET io_offset = 0x23
.SET porta = io_offset + 2
.CSEG                              ; Start code segment
        clr      r2                ; Clear register 2
        out      porta,r2          ; Write to Port A
```

```
…
.SET io_offset = 0x60
```


Assembler Directives

Directive	Description	Examples
.include	Include a file as part of the source code	**.include "iodefs"**
.exit	The EXIT directive tells the assembler to stop assembling the file	**…** **.exit**
.org	Set the location counter of the current segment to a specific value	**.cseg ; set flash location** **.org 0x002A** **…** **.dseg ; set SRAM location** **.org 64**
.device	Define which device to assemble for. • If this directive is used, a warning is issued if an instruction not supported by the specified device occurs in the code. • If the size of the Code Segment or EEPROM Segment is larger than supported by the specified device, a warning is issued.	**.device AT90S1200**

Example

Example:

```
.DEVICE AT90S1200              ; Use the AT90S1200
.CSEG
        push    r30            ; This statement will generate
                               ; a warning since the
                               ; specified device does not
                               ; have this instruction
```


.DEVICE – Define which device to assemble for

Example: .INCLUDE “tn45def.inc”

```
;***** THIS IS A MACHINE GENERATED FILE - DO NOT EDIT ********************
;***** Generated: 2004-10-19 10:12 ******* Source: ATtiny45.xml **********
;************************************************************************
;* A P P L I C A T I O N   N O T E   F O R   T H E   A V R   F A M I L Y
;*
;* Number          : AVR000
;* File Name       : "tn45def.inc"
;* Title           : Register/Bit Definitions for the ATtiny45
;* Date            : 2004-10-19
;* Version         : 2.02
;* Support E-mail  : avr@atmel.com
;* Target MCU      : ATtiny45
;*
;* DESCRIPTION
;* When including this file in the assembly program file, all I/O register
;* names and I/O register bit names appearing in the data book can be used.
;* In addition, the six registers forming the three data pointers X, Y and
;* Z have been assigned names XL - ZH. Highest RAM address for Internal
;* SRAM is also defined
;*
;* The Register names are represented by their hexadecimal address.
;*
;* The Register Bit names are represented by their bit number (0-7).
;*
;* Please observe the difference in using the bit names with instructions
;* such as "sbr"/"cbr" (set/clear bit in register) and "sbrs"/"sbrc"
;* (skip if bit in register set/cleared). The following example illustrates
;* this:
;*
;* in    r16,PORTB             ;read PORTB latch
;* sbr   r16,(1<<PB6)+(1<<PB5) ;set PB6 and PB5 (use masks, not bit#)
;* out   PORTB,r16             ;output to PORTB
;*
;* in    r16,TIFR              ;read the Timer Interrupt Flag Register
;* sbrc  r16,TOV0              ;test the overflow flag (use bit#)
;* rjmp  TOV0_is_set           ;jump if set
;* ...                         ;otherwise do something else
;************************************************************************

; ***** SPECIFY DEVICE ***************************************************
.device ATtiny45

; ***** I/O REGISTER DEFINITIONS *****************************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    SREG    = 0x3f
.equ    SPH     = 0x3e
.equ    SPL     = 0x3d
.equ    GIMSK   = 0x3b
.equ    GIFR    = 0x3a
.equ    TIMSK   = 0x39
.equ    TIFR    = 0x38
.equ    SPMCSR  = 0x37
...
```

Example - .include

Contents of iodefs.asm

```
                                ; iodefs.asm:
.EQU    sreg=0x3f               ; Status register
.EQU    sphigh=0x3e             ; Stack pointer high
.EQU    splow=0x3d              ; Stack pointer low
                                ; incdemo.asm
```

Include this in another ASM program, and one can use symbols such as sreg rather than 0x3f:

```
.INCLUDE "iodefs.asm"           ; Include I/O definitions
        in    r0,sreg           ; Read status register
```


Example - .org

Example:

```
.DSEG                           ; Start data segment
.ORG 0x67                       ; Set SRAM address to hex 67
        variable:.BYTE 1        ; Reserve a byte at SRAM
                                ; adr.67H
.ESEG                           ; Start EEPROM Segment
.ORG 0x20                       ; Set EEPROM location
                                ; counter
        eevar:   .DW 0xfeff     ; Initialize one word
.CSEG
.ORG 0x10                       ; Set Program Counter to hex
                                ; 10
        mov      r0,r1          ; Do something
```

Example - Macro

.MACRO – Begin macro

.ENDMACRO – Ends macro

.LISTMAC – Show macros expanded in list file

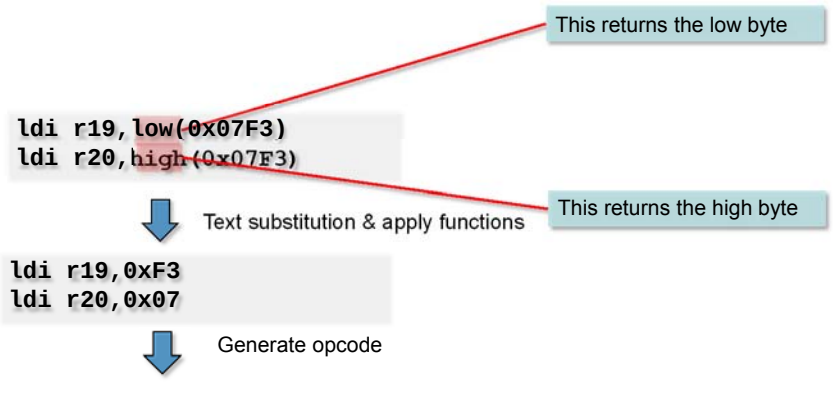
```
.LISTMAC

.MACRO SUBI16                ; Start macro definition
   subi @1,low(@0)           ; Subtract low byte
   sbci @2,high(@0)          ; Subtract high byte
.ENDMACRO                    ; End macro definition

.CSEG
   SUBI16 0x1234,r16,r17     ; Sub.0x1234 from r17:r16
```

Text substitution/macro expansion

```
subi r16 , low ( 0x1234 )
sbci r17 , high ( 0x1234 )
```

Generate opcode

Example - Assembler Functions



Assembler Functions

LOW and HIGH are often used, others less frequently, and we will not cover these in this course

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2^expression
- LOG2(expression) returns the integer part of log2(expression)

Assembler Operators

Assembler functions and operators are related to assembler directives.

The functions and operators are not translated directly into opcode.

Instead, they are used to do calculations on the assembly language source file.

```
.EQU c1 = 0b11011010
.equ c2 = 0b10101010

ldi r18,c1^c2
```

This is the Assembler bit-wise exclusive or (XOR) operator.

Text substitution & apply functions

```
ldi r18,0x70
```

Generate opcode

Some Assembler Operators

Multiplication

Symbol: *

Description: Binary operator which returns the product of two expressions

Precedence: 13

Example:
```
ldi    r30,label*2     ; Load r30 with label*2
```

Division

Symbol: /

Description: Binary operator which returns the integer quotient of the left expression divided by the right expression

Precedence: 13

Example:
```
ldi    r30,label/2     ; Load r30 with label/2
```


Some Assembler Operators

Addition

Symbol: +

Description: Binary operator which returns the sum of two expressions

Precedence: 12

Example:
```
ldi    r30,c1+c2       ; Load r30 with c1+c2
```

Subtraction

Symbol: -

Description: Binary operator which returns the left expression minus the right expression

Precedence: 12

Example:
```
ldi    r17,c1-c2       ;Load r17 with c1-c2
```

Some Assembler Operators

Shift left

Symbol: <<

Description: Binary operator which returns the left expression shifted left a number of times given by the right expression

Precedence: 11

Example:
```
ldi   r17,1<<bitmask ;Load r17 with 1 shifted
                     ;left bitmask times
```

Shift right

Symbol: >>

Description: Binary operator which returns the left expression shifted right a number of times given by the right expression.

Precedence: 11

Example:
```
ldi   r17,c1>>c2     ;Load r17 with c1 shifted
                     ;right c2 times
```


Some Assembler Operators

Bitwise AND

Symbol: &

Description: Binary operator which returns the bitwise And between two expressions

Precedence: 8

Example:
```
ldi   r18,High(c1&c2)   ;Load r18 with an expression
```

Bitwise OR

Symbol: |

Description: Binary operator which returns the bitwise Or between two expressions

Precedence: 6

Example:
```
ldi   r18,Low(c1|c2)    ;Load r18 with an expression
```

Some Assembler Operators

Logical AND

Symbol: &&

Description: Binary operator which returns 1 if the expressions are both nonzero, 0 otherwise

Precedence: 5

Example:
```
ldi    r18,Low(c1&&c2)    ;Load r18 with an expression
```

Logical OR

Symbol: ||

Description: Binary operator which returns 1 if one or both of the expressions are nonzero, 0 otherwise

Precedence: 4

Example:
```
ldi    r18,Low(c1||c2)    ;Load r18 with an expression
```


Some Assembler Operators

Logical NOT

Symbol: !

Description: Unary operator which returns 1 if the expression was zero, and returns 0 if the expression was nonzero

Precedence: 14

Example:
```
ldi    r16,!0xf0       ; Load r16 with 0x00
```

Bitwise NOT

Symbol: ~

Description: Unary operator which returns the input expression with all bits inverted

Precedence: 14

Example:
```
ldi    r16,~0xf0       ; Load r16 with 0x0f
```

Some Assembler Operators

Not Equal

Symbol: !=

Description: Binary operator which returns 1 if the signed expression to the left is Not Equal to the signed expression to the right, 0 otherwise

Precedence: 9

Example:
```
.SET flag=(c1!=c2)        ;Set flag to 1 or 0
```

Equal

Symbol: ==

Description: Binary operator which returns 1 if the signed expression to the left is Equal to the signed expression to the right, 0 otherwise

Precedence: 9

Example:
```
andi   r19,bitmask*(c1==c2)+1      ;And r19 with
                                   ;an expression
```


Some Assembler Operators

Less Than

Symbol: <

Description: Binary operator which returns 1 if the signed expression to the left is Less than the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori    r18,bitmask*(c1<c2)+1       ;Or r18 with
                                   ;an expression
```

Greater Than

Symbol: >

Description: Binary operator which returns 1 if the signed expression to the left is Greater than the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori    r18,bitmask*(c1>c2)+1       ;Or r18 with
                                   ;an expression
```

Assembler Preprocessor

The AVRASM2 preprocessor is modeled after the C preprocessor, with some exceptions (see AVRASM2 documentation).

#define	#if	#pragma	Operators:
#elif	#ifdef	#undef	# (stringification
#else	#ifndef	#warning	## (concatenation)
#endif	#include	# (empty directive)	
#error	#message		

```
.equ c1 = 0b11011010
.equ c2 = 0b10101010

#define DEBUG

#ifdef DEBUG
   ldi r18,low(c1|c2)
   rcall delay
#else
   ldi r18,low(c1^c2)
#endif
```

As with the C preprocessor, the AVRASM2 preprocessor allows conditional processing.

For example, one can build a special debug version as shown in the snippet.

Example

Write an AVR assembler program that configures PB4, PB1, and PB0 as output and PB5, PB3, and PB2 as input. Make sure that your intensions are clearly visible in the program!

DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x17	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Solution:

```
ldi R16, (1<<DDB4) | (1<<DDB1) | (1<<DDB0)
out DDRB, R16
```

Needs the correct µC specific definition file (.include)!

Assembler Directives - Example

```
.DEF temp=r23
.CSEG
.ORG 0
ldi R28,0x35
ldi R29,0x01
ld R22,Y
inc R23
st Y+,temp
rjmp l1
.DW 0x2014, $20AA
l1:
inc R23
st Y+, r23

.ESEG
.ORG 10
T1: .DB 0b10100111, 12, 0xFE
V1: .BYTE 10
V2: .BYTE 1
```

```
.DSEG
.ORG 0x100
array1: .BYTE 10
b: .BYTE 1
x2: .BYTE 1
table3: .BYTE 22

.ESEG
T2: .DW 0b1100100110111110, 142, 0x23FE
V3: .BYTE 1

.CSEG
inc R23
l2: TST R23
```

What is the address of label "l2"?

Answer: l2 → 0x000B

… EOL