Microprocessors & Interfacing

AVR Programming (II)

Lecturer : Annie Guo

Lecture Overview

- Assembly program structure
  - Assembler directive
  - Assembler expression
  - Macro

Assembly Program Structure

- An assembly program basically consists of
  - Assembler directives
    - E.g. ***.def temp = r15***
  - Executable instructions
    - E.g. ***add r1, r2***
- A line in an assembly program takes one of the following forms :
  - [label:] directive [operands] [comment]
  - [label:] instruction [operands] [comment]
  - Comment
  - Empty line

Note: [ ] indicates optional

Assembly Program Structure (cont.)

- The label for an instruction or a data item in the memory is **associated with the memory address** of that instruction or that data item.
- All instructions are not case sensitive
  - **“add” is same as “ADD”**
  - **“.def” is same as “.DEF”**

Comments

- A comment line has the following form:

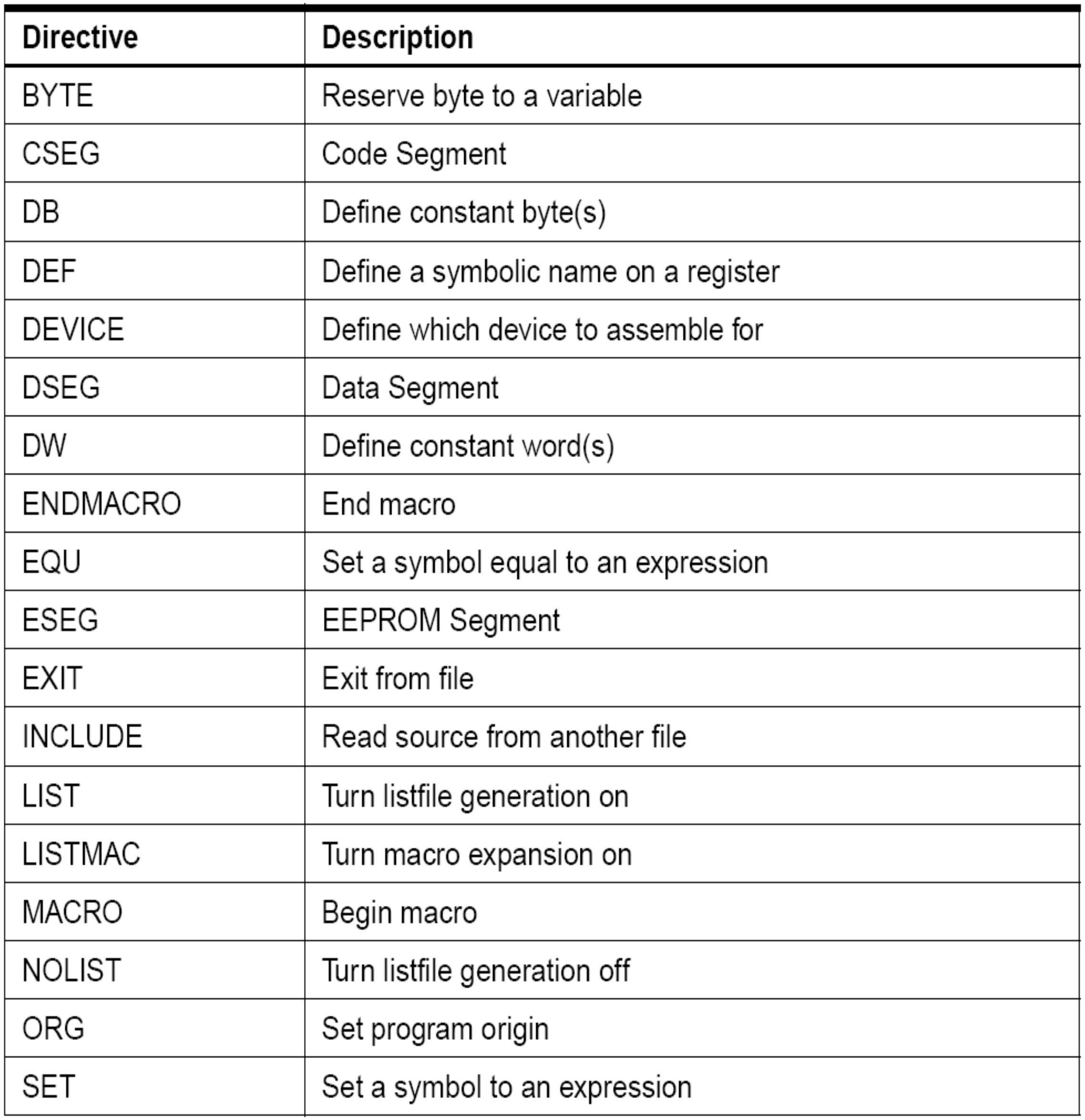
  ;[text]

  Items within the brackets are optional
- The text between the comment-delimiter(;) and the end of line (EOL) is ignored by the assembler.

Assembly Directives

- Assembly directives are instructions to the assembler. They are used for a number of purposes:
  - For symbol definition
    - For readability and maintainability
    - All symbols used in a program will be replaced by the real values associated with the symbol during assembling
    - E.g. .def, .set
  - For program and data organization
    - E.g. .org, .cseg, .dseg
  - For data/variable memory allocation
    - E.g. .db
  - For others

Typical AVR Assembler directives



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

NOTE: All directives must be preceded by **a period, '.'**

Directives for Symbol Definition

.def

- Define a symbol/alias for a **register**

```
.def       symbol = register
```

- E.g.

```
.def  temp = r17
```

  - Symbol *temp* can be used for r17 anywhere in the program after the definition

Directives for Symbol Definitions (cont.)

.equ

- Define a symbol for a **value**

```
.equ        symbol = expression
```

  - Non-redefinable. Once set, the symbol cannot be later redefined to other value in the program

- E.g.

```
.equ  length = 2
```

  - Symbol *length* with value 2 can be used anywhere in the program after the definition

Directives for Symbol Definitions (cont.)

.set

- Define a symbols for a **value**

```
.set        symbol = expression
```

  - **Re-definable**. The symbol can be changed later to represent other value in the program.

- E.g.

```
.set  input = 5
```

  - Symbol *input* with value 5 can be used anywhere in the program after this definition and before its redefinition.

Program/Data Memory Organization

- AVR has three different memories
  - – Data memory
  - – Program memory
  - – EPROM memory (not covered in this course)
- The memories are corresponding to memory segments to the assembler:
  - – Data segment
  - – Program segment (or Code segment)

•Segment here is referred to as a memory space

Program/Data Memory Organization Directives

- Memory segment directives specify which memory to use
  - **.dseg**
    - Data memory
  - **.cseg**
    - Code/Program memory
- The default segment is cseg
- The **.org** directive specifies the start address for the related code/data to be saved

- The default start location is hardware oriented.

Example

```
           .dseg                 ; Start the data segment
           .org    0x0300        ; from address 0x0300,
                                 ; default start location is 0x0200

vartab:  .byte   4               ; Reserve 4 bytes in SRAM
                                 ; from address 0x0300

           .cseg                 ; Start the code segment
                                 ; default start location is 0x00000

const:     .dw    10,  0x10, 0b10, -1
                                  ; Save 10, 16, 2, -1 in program
                                  ; memory, each value takes
                                  ; 2 bytes.

           mov    r1, r0         ; Do something
```

Data/Variable Memory Allocation Directives

- Specify the memory locations/sizes for
  - Constants
    - In program memory
  - Variables
    - In data memory
- All directives must start with a label so that the related data/variables can be accessed later.

Directives for Constants

- Store data in **program memory**
  - **.db**
    - Store **byte** constants in program memory

```
label:  .db    expr1, expr2, …
```

      - ***expr**** **is a byte constant**
  - **.dw**
    - Store **word** (16-bit) constants in program memory
    - **little endian** rule is used

小字节序、低字节序）,即低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

```
label:  .dw    expr1, expr2, …
```

      - ***expr**** **is a word constant**

Directives for Variables

- Reserve bytes in **data memory**
  - **.byte**
    - Reserve a number of bytes for a variable

```
Label:  .byte   expr
```

    - ***expr*** **is the number of bytes to be reserved.**

Other Directives

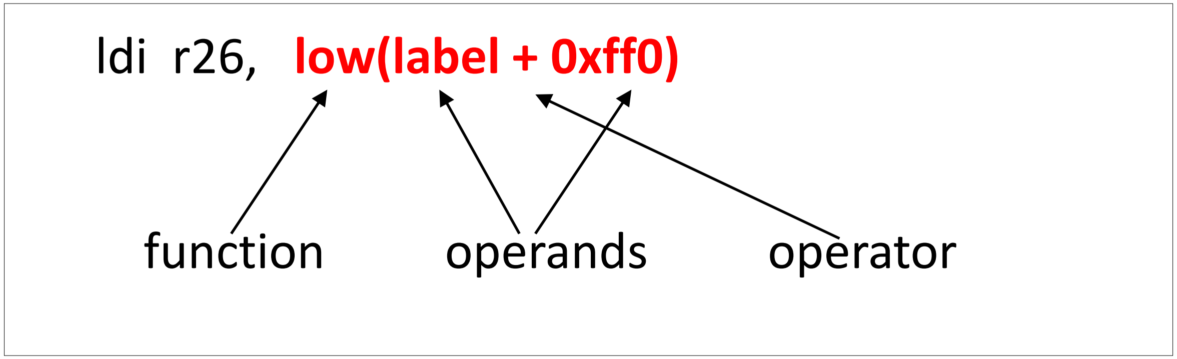
- Include a file
  - **.include** “m2560def.inc”
- Stop processing assembly file
  - **.exit**
- Define macro
  - **.macro**
  - **.endmacro**
  - Will be discussed in detail later

Assembler Expressions

- In the assembly program, you can use expressions for values.
- **During assembling**, the assembler evaluates each expression and replaces the expression with the calculated value.

Assembler Expressions (cont.)

- The expressions are in a form similar to normal math expressions
  - Consisting of operands, operators and functions. All expressions can be of a value up to 32 bits.
- Example



```
ldi  r26,   low(label + 0xff0)
```

function

operands

operator

Operands in Assembler Expression

- Operands can be any of the following:
  - User defined labels
    - associated with memory addresses
  - User defined variables
    - defined by the 'set' directive
  - User defined constants
    - defined by the 'equ' directive
  - Integer constants
    - can be in several formats, including
      - decimal (default): e.g. 10, 255
      - hexadecimal (two notations): e.g. 0x0a, $0a, 0xff, $ff
      - binary: e.g. 0b00001010, 0b11111111
  - PC
    - Program Counter value.

Operators in Assembler Expression

Same meanings as in C

Symbol	Description
!	Logical Not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise And
^	Bitwise Xor
|	Bitwise Or
&&	Logical And
||	Logical Or

Functions in Assembler Expression

- LOW(expression)
  - Returns the low byte of an expression
- HIGH(expression)
  - Returns the second (low) byte of an expression
- BYTE2(expression)
  - 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 low word (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 to the power of expression
- LOG2(expression):
  - Returns the integer part of log2(expression)

Examples of Assembler Expression

```
; Example 1:

          ldi  r17, 1<<5   ; load r17 with 1 left-shifted by 5 bits
```

Examples of Assembler Expression

```
; Example 2: compare r21:r20 with 3167

            ldi   r16, high(3167)
            ldi   r17, low(3167)
            cp   r20,  r17
            cpc  r21, r16                          带进位比较
            brlt  case1    小于转（带符号）
            …
case1:    inc  r10
```

Data/Variables Implementation

- With the assembler directives, you can implement/translate data/variables into machine level descriptions
  - – See some examples in the next a few slides.

Remarks

- Data have scope and duration in the program
- Data have types and structures
- Those features determine where and how to store data in memory.
- Constants are usually stored in the non-volatile memory and variables are allocated in SRAM memory.
- In this lecture, we will only take a look at how to implement basic data type.
  - Implementation of advanced data structures/variables will be covered later.

Example 1

- Translate the following C variables. Assume each integer takes four bytes.

```
int  a;
unsigned int  b;
char  c;
char*  d;
```

Example 1: Solution

- Translate the following variables. Assume each integer takes four bytes.

```
.dseg                  ; in data memory

.org 0x200             ; start from address 0x200

a:   .byte  4          ; 4 byte integer
b:   .byte  4          ; 4 byte unsigned integer
c:   .byte   1         ; 1 character
d:   .byte  2          ; address pointing to the string
```

  - All variables are allocated in data memory (SRAM)
  - Labels are given the same names as the variable for convenience and readability.

- FLASH – Code memory
- ASCII is used for character

Example 2

- Translate the following C constants and variables.

C code:

```
int  a;
const char b[ ] = “COMP9032”;
const int c = 9032;
```

Assembly code:

```
.dseg
a:  .byte 4

.cseg
;b:  .db  ‘C’, ‘O’, ‘M’, ‘P’, ‘9’, ‘0’, ‘3’, ‘2’, 0
b:  .db  “COMP9032”, 0
c:  .dw  9032
```

  - All variables are in SRAM and constants are in FLASH

Example 2 (cont.)

- Program memory mapping
  - In the program memory, data are packed in words. If only a single byte left, that byte is stored in the first (left) byte and the second (right) byte is filled with 0, as highlighted in the example.

	low addr	high addr
0x0000	‘C’	‘O’
0x0001	‘M’	‘P’
0x0002	‘9’	‘0’
0x0003	‘3’	‘2’
0x0004	0	0
0x0005	0x48	0x23

Hex values

43	4F
4D	50
39	30
33	32
0	0
48	23

Example 3

- Translate variables with structured data type

```
struct STUDENT_RECORD
{
            int  student_ID;
            char  name[20];
            char  WAM;
};

typedef struct STUDENT_RECORD student;

student s1;
student s2;
```

Example 3 : Solution

- Translate variables with structured data type

```
.set      student_ID=0
.set      name = student_ID+4
.set      WAM = name + 20
.set      STUDENT_RECORD_SIZE = WAM + 1

.dseg
s1:       .BYTE    STUDENT_RECORD_SIZE
s2:       .BYTE    STUDENT_RECORD_SIZE
```

Example 4

- Translate variables with structured data type
  - with initialization

```
struct STUDENT_RECORD
{
            int  student_ID;
            char  name[20];
            char  WAM;
};

typedef struct STUDENT_RECORD student;

struct student  s1 = {123456, “John Smith”, 75};
struct student  s2;
```

Example 4: Solution

- Translate variables with structured data type

```
.set        student_ID=0
.set        name = student_ID+4
.set        WAM = name + 20
.set        STUDENT_RECORD_SIZE = WAM + 1

.cseg
s1_value:     .dw    LWRD(123456)
              .dw    HWRD(123456)
              .db    “John Smith        ”, 0
              .db    75
.dseg
s1:         .byte    STUDENT_RECORD_SIZE
s2:         .byte    STUDENT_RECORD_SIZE

; copy the data from instruction memory to s1
…
```

Remarks

- The constant values for initialization are usually stored in the program memory in order to keep the values when power is off.
- The variables will be populated with the initial values when the program is started.

Macro

- Sometimes, a sequence of instructions in an assembly program need to be repeated several times
- Macros help programmers to write code efficiently and nicely
  - Write/define a section of code once and reuse it
    - Neat representation
  - Like an inline function in C
    - When assembled, the macro is expanded at the place it is used

Directives for Macro

.macro

- Tells the assembler that this is the start of a macro
- Takes the macro name and (implicitly) parameters
  - Up to 10 parameters
    - Which are referenced by @0, …@9 in the macro definition body

.endmacro

- Specifies the end of a macro definition.

```
.macro macro_name
              ; macro body
.endmacro
```

Macro (cont.)

- Macro definition structure:

```
.macro macro_name
             ; macro body
.endmacro
```

- Usage

```
macro_name   [para0, para1, …,para9]
```

Example 1

- Swapping two memory data

```
.macro swap2
            lds r2, @0            ; load data from provided
            lds r3, @1            ; two locations
            sts @1, r2            ; interchange the data and
            sts @0, r3            ; store data back
.endmacro

swap2 a, b                        ; a is @0, b is @1.
swap2 c, d                        ; c is @0, d is @1.
```

Example 2

- Register bit copy
  - copy a bit from one register to a bit of another register

```
; Copy bit @1 of register @0
; to bit @3 of register @2

.macro  bitcopy
            bst @0, @1
            bld @2, @3
.endmacro

bitcopy r4, 2, r5, 3
bitcopy r5, 4, r7, 6
```

Reading Material

- Cady “Microcontrollers and Microprocessors”, Chapter 6 for assembly programming style.
- User’s guide to AVR assembler
  - This guide is a part of the on-line documentations accompanied with AVR Studio. Click help in AVR Studio.

Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
    - Arithmetic and logic instructions
        - clr
        - inc, dec
    - Data transfer instructions
        - movw
        - sts, lds
        - lpm
        - bst, bld
    - Program control
        - jmp
        - sbrs, sbrc

2. Complete Quiz 2