Microprocessors & Interfacing

AVR Programming (II)

Lecturer : Annie Guo

Lecture Overview

- Assembly program structure
  - Assembler directive
  - Assembler expression
  - Macro
- Memory access
- Assembly process
  - First pass
  - Second pass

Assembly Program Structure

- An assembly program basically consists of
  - Assembler directives
    - E.g. ***.def temp = r15***
  - Executable instructions
    - E.g. ***add r1, r2***
- An input 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.

Example

```
; The program performs
; 2-byte addition: sum=a+b;

      .def   a_high = r2;
      .def   a_low = r1;
      .def   b_high = r4;
      .def   b_low = r3;
      .def   sum_high = r6;
      .def   sum_low = r5;

      mov    sum_low, a_low
      mov    sum_high, a_high
      add    sum_low, b_low
      adc    sum_high, b_high
end:  rjmp end
```

Two comment lines

Empty line

Six assembler directives

Five executable instructions

Assembly Directives

- Assembly directives are instructions to the assembler. They are used for a number of purposes:
  - For symbol definitions
    - For readability and maintainability
    - All symbols used in a program will be replaced by the real values 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 Definitions

- **.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 symbols for **values**

```
.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 symbols for **values**

```
.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
- The three memories are corresponding to three memory segments to the assembler:
  - Data segment
  - Program segment (or Code segment)
  - EEPROM segment

•Segment here is referred to as a memory space

Program/Data Memory Organization Directives

- Memory segment directive specifies which physical memory to use
  - **.dseg**
    - Data memory
  - **.cseg**
    - Code/Program memory
  - **.eseg**
    - EPROM 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
                                ; Write 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/EEPROM 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**/**EEPROM** **memory**
  - **.db**
    - Store **<u>byte</u>** constants in program memory

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

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

```
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. <u>0x</u>0a, $0a, 0xff, $ff
      - binary: e.g. <u>0b</u>00001010, 0b11111111
      - octal (leading zero): e.g. <u>0</u>10, 077
  - PC
    - Program Counter value.

Operators in Assembler Expression

Same meanings as in C

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

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

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

Examples

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

            ldi    r16, high(3167)
            cpi    r20, low(3167)
            cpc    r21, r16
            brlt   case1
            ...
case1:      inc    r10
```

Data/Variables Implementation

- With the assembler directives, you can implement/translate data/variables into machine level descriptions

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 name 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.

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          offset
.set      name = student_ID+4
.set      WAM = name + 20
.set      STUDENT_RECORD_SIZE = WAM + 1

.dseg                          25 = 4 + 2 + 1
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    name[20]
            .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
  - Type/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 (cont.)

- Macro definition structure:

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

- Usage

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

Example 1

- Swapping memory data *p*, *q* for a data shuffling operation
  - assume the two data are in memory location *p* and *q* respectively

Without macro

```
lds r2, p
lds r3, q
sts q, r2
sts p, r3
```

With macro

```
.macro swap1
        lds r2, p     ; load data
        lds r3, q     ; from p, q
        sts q, r2     ; store data
        sts p, r3     ; to q, p
.endmacro

swap1
```

Example 2

- Swapping any 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 3

- 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
```

Memory Access Operations

- Access to data memory
  - Using instructions
    - ld, lds, st, sts
- Access to program memory
  - Using instructions
    - lpm
    - spm
      - Not covered in this course
  - Most of time, that we access the program memory is to load data

Load Program Memory Instruction

- Syntax: ***lpm Rd, Z***
- Operands: $Rd \in \{r0, r1, \ldots, r31\}$
- Operation: $Rd \leftarrow (Z)$

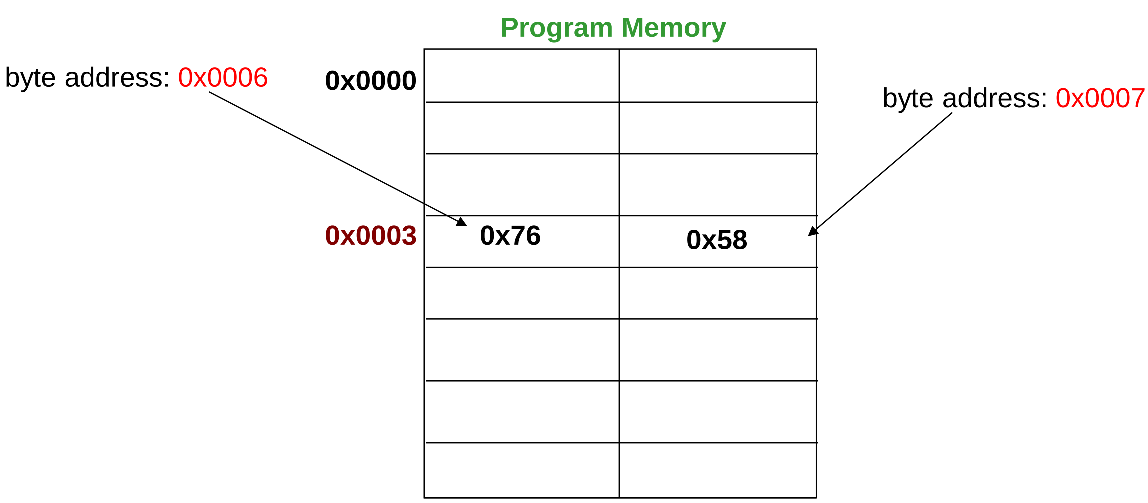
- Words: 1
- Cycles: 3

Load Data From Program Memory

- The address label in the program memory is ***word address***
  - Used by the PC register
- To access constant data in the program memory with *lpm*, ***byte address*** should be used.
- Address register, Z, is used to point bytes in the program memory

Byte Address vs Word Address

- First-byte-address (in a word) = 2 * word-address
- Second-byte-address (in a word) = 2 * word-address +1



Example

```
.include "m2560def.inc"   ; include definition for Z

                 = table_1 * 2
ldi ZH, high(Table_1<<1)  ; initialize Z
ldi ZL, low(Table_1<<1)

lpm r16, Z                ; load constant from the program
                          ; memory pointed to by Z (r31:r30)

table_1:          word address
        .dw 0x5876        ; 0x76 is the value when Z_LSB = 0
                          ; 0x58 is the value when Z_LSB = 1
```

Complete Example 1

- Copy data from Program memory to Data memory

Complete Example 1 (cont.)

- C description

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

typedef struct STUDENT_RECORD student;

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

Complete Example 1 (cont.)

- Assembly translation

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

.cseg
start:      ldi zh, high(s1_value<<1)       ; pointer to student record
            ldi zl, low(s1_value<<1)        ; value in the program memory

            ldi yh, high(s1)                ; pointer to student record holder
            ldi yl, low(s1)                 ; in the data memory

            clr r16
```

Complete Example 1 (cont.)

- Assembly translation (cont.)

```
load:
                cpi r16, STUDENT_RECORD_SIZE
                brge end
                lpm r10, z+
                st  y+, r10
                inc r16
                rjmp load
end:
                rjmp end

s1_value:       .dw     LWRD(123456)
                .dw     HWRD(123456)
                .db     "John Smith        ", 0
                .db     75
.dseg
.org 0x200
s1:     .byte   STUDENT_RECORD_SIZE
```

Assembly

- Assembly programs need to be converted to machine code before execution
  - This translation/conversion from assembly program to machine code is called ***assembly*** and is done by the ***assembler***
- There are two general steps in the assembly processes:
  - Pass one
  - Pass two

Two Passes in Assembly

- Pass One
  - Lexical and syntax analysis: checking for syntax errors
  - Expand macro calls
  - Record all the symbols (labels etc) in a symbol table
- Pass Two
  - Use the symbol table to substitute the values for the symbols and evaluate functions.
  - Assemble each instruction
    - i.e. generate machine code

Example

Assembly program

```
.equ     bound = 5

         clr r16
loop:
         cpi r16, bound
         brlo end
         inc r16
         rjmp loop
end:
         rjmp end
```

Symbol table

Symbol	Value
bound	5
loop	1
end	5

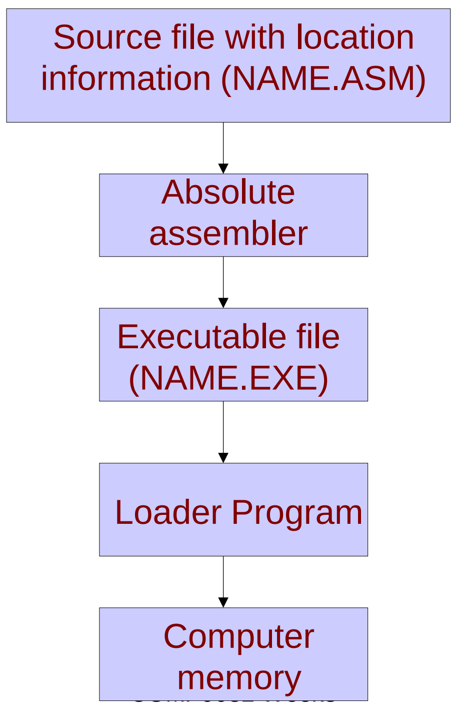
Example (cont.)

Address	Code	Assembly statement	
00000000:	2700	clr	r16
00000001:	3005	cpi	r16,0x05
00000002:	F010	brlo	PC+0x03
00000003:	9503	inc	r16
00000004:	CFFC	rjmp	PC-0x0003
00000005:	CFFF	rjmp	PC-0x0000

Absolute Assembly

- A type of assembly process.
  - Can only be used for the source file that contains all the source code of the program
- Programmers use .org to tell the assembler the starting address of a segment (data segment or code segment)
- Whenever any change is made in the source program, all code must be assembled.
- A loader transfers an **executable file** (machine code) to the target system.

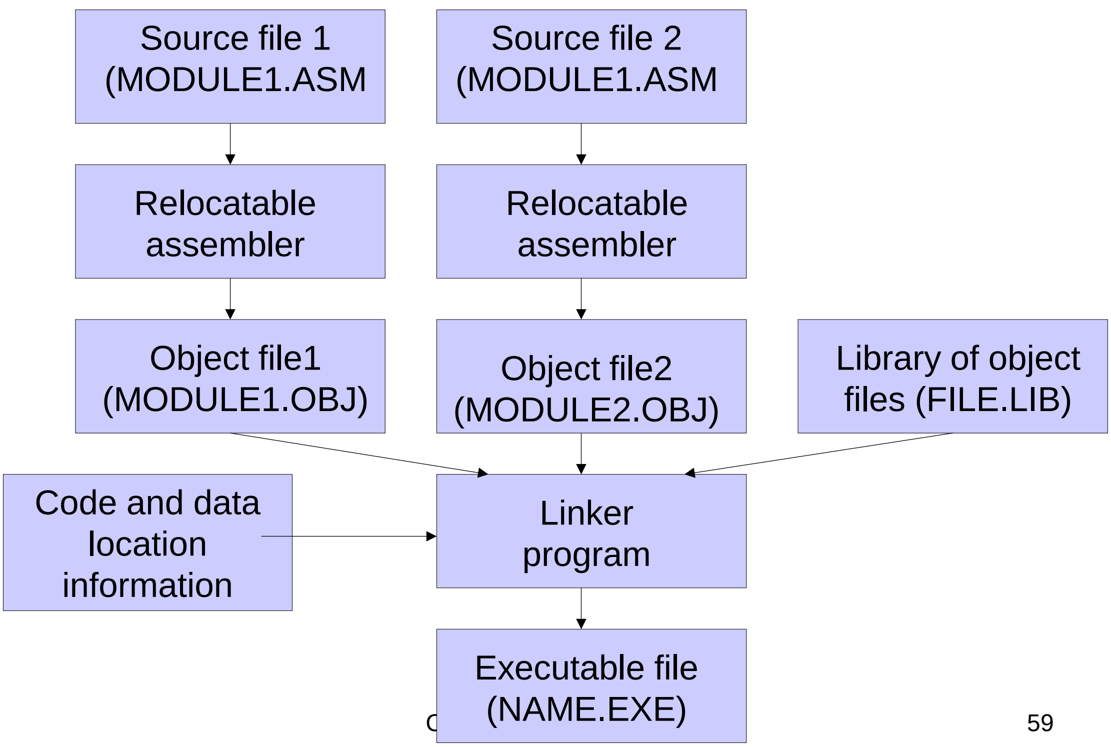
Absolute Assembly - workflow



Relocatable Assembly

- Another type of assembly process.
- Each source file can be assembled separately
- Each file is assembled into **an object file** where some addresses may not be resolved
- A linker program is needed to resolve all unresolved addresses and make all object files into a single executable file

Relocatable Assembly - workflow



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

Homework

2. Design a checking strategy that can find the endianness of AVR machine.

Homework

3. Convert lowercase to uppercase for a string (for example, “hello”)
   - The string is stored in the program memory
   - The resulting string after conversion is stored in the data memory.
     - In ASCII, uppercase letter + 32 = lowercase letter
       - e.g. ‘A’+32=‘a’

Advertisement