Assembly Programming (II)

Lecturer: Sri Parameswaran
Notes by: Annie Guo
Lecture overview

- Assembly program structure
  - Assembler directives
  - Assembler expressions
  - Macro
- Memory access
- Assembly process
  - First pass
  - Second pass
Assembly program structure

- An assembly program basically consists of:
  - Assembler directives
    - E.g. `.equ constant = 19`
  - 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
Assembly program structure (cont.)

- The label for an instruction is associated with the memory location address of that instruction.

- All instructions are not case sensitive
  - "add" is same as "ADD"
  - ".equ" is same as ".EQU"
Example

; The program performs
; 3-way addition: a+b+c;

.equ A = 4
.equ B = 8
.equ C = 9

ldi r16, A
ldi r17, B
ldi r18, C
add r16, r17
add r16, r18

Two comment lines
Empty line
Three assembler directives
Five executable instructions
Comments

- A comment 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

- Instructions to the assembler are created 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 when assembling
    - E.g. `.equ`, `.set`
  - For program and data organization
    - E.g. `.org`, `.cseg`, `.dseg`
  - For data/variable memory allocation
    - E.g. `.db`, `.dw`
  - For others
<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>Reserve byte to a variable</td>
</tr>
<tr>
<td>CSEG</td>
<td>Code Segment</td>
</tr>
<tr>
<td>DB</td>
<td>Define constant byte(s)</td>
</tr>
<tr>
<td>DEF</td>
<td>Define a symbolic name on a register</td>
</tr>
<tr>
<td>DEVICE</td>
<td>Define which device to assemble for</td>
</tr>
<tr>
<td>DSEG</td>
<td>Data Segment</td>
</tr>
<tr>
<td>DW</td>
<td>Define constant word(s)</td>
</tr>
<tr>
<td>ENDMACRO</td>
<td>End macro</td>
</tr>
<tr>
<td>EQU</td>
<td>Set a symbol equal to an expression</td>
</tr>
<tr>
<td>ESEG</td>
<td>EEPROM Segment</td>
</tr>
<tr>
<td>EXIT</td>
<td>Exit from file</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Read source from another file</td>
</tr>
<tr>
<td>LIST</td>
<td>Turn listfile generation on</td>
</tr>
<tr>
<td>LISTMAC</td>
<td>Turn macro expansion on</td>
</tr>
<tr>
<td>MACRO</td>
<td>Begin macro</td>
</tr>
<tr>
<td>NOLIST</td>
<td>Turn listfile generation off</td>
</tr>
<tr>
<td>ORG</td>
<td>Set program origin</td>
</tr>
<tr>
<td>SET</td>
<td>Set a symbol to an expression</td>
</tr>
</tbody>
</table>

NOTE: All directives must be preceded by a period
Directives for symbol definitions (cont.)

- .equ
  - Define symbols for **values**
    
    
    `.equ symbol = expression`

    - Non-redefinable. Once set, the symbol cannot be redefined to other value later 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 to represent other values later 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.
Directives for symbol definitions

- **.def**
  - Define an alias for a **register**
    
    ```
    .def symbol = register
    ```
  - **E.g.**
    ```
    .def ZL = r30
    ```
    - Symbol temp can be used instead of r30 anywhere in the program after the definition
Program/data memory organization

- AVR has three different memories
  - Data memory
  - Program memory
  - EEPROM memory
- The three memories correspond to three memory segments to the assembler:
  - Data segment (or RAM)
  - Program segment (or Code segment, Flash)
  - EEPROM segment
Program/data memory organization directives

- Memory segment directives specify which memory segment to use
  - .dseg
    - Data segment
  - .cseg
    - Code segment
  - .eseg
    - EEPROM segment

- The .org directive specifies the start address to store the related program/data.
Example

.dseg ; Start data segment

vartab: .byte 4 ; Reserve 4 bytes in SRAM
; (Will be at 0x200 on the atmega2560)

.cseg ; Start code segment
; default start location is 0x0000

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/variable can be accessed later.
Directives for Constants

- Store data in program/EEPROM memory
  - .db
    - Store `byte` constants in program/EEPROM memory
      
      ```
      Label: .db expr1, expr2, ...
      ```
      - `expr*` is a byte constant value or string literal
  - .dw
    - Store `word` (16-bit) constants in program/EEPROM memory
    - **Little endian** rule is used
      
      ```
      Label: .dw expr1, expr2, ...
      ```
      - `expr*` is a word constant value
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 the assembly file
  - `.exit`
- Begin and end macro definition
  - `.macro`
  - `.endmacro`
  - Will be discussed in detail later
Implement data/variables

- With those directives, you can implement/translate data/variables into machine level descriptions
- An example of translation by WINAVR is given in the next slide.
Sample C program

```c
// global variables:
const char g_course[] = "COMP";
char* g_inputCourse = "COMP";
char g_a;
static char g_b;

int main(void) {
    // local variables:
    const char course[] = "COMP9032";
    char* inputCourse = "COMP9032";
    char a;
    static char b;
    char i;
    char isCOMP9032 = 1;

    for(i=0; i<9; i++){
        if (inputCourse[i] != course[i]) {
            isCOMP9032 = 0;
            i = 9;
        }
    }
    return 0;
}
```
Memory mapping after build and run

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_course</td>
<td>[... ]</td>
<td>const char[5]</td>
<td>0x0100 [SRAM]</td>
</tr>
<tr>
<td>[0x00]</td>
<td>67 'C'</td>
<td>const char</td>
<td>0x0100 [SRAM]</td>
</tr>
<tr>
<td>[0x01]</td>
<td>79 'O'</td>
<td>const char</td>
<td>0x0101 [SRAM]</td>
</tr>
<tr>
<td>[0x02]</td>
<td>77 'M'</td>
<td>const char</td>
<td>0x0102 [SRAM]</td>
</tr>
<tr>
<td>[0x03]</td>
<td>80 'P'</td>
<td>const char</td>
<td>0x0103 [SRAM]</td>
</tr>
<tr>
<td>[0x04]</td>
<td>0 ''</td>
<td>const char</td>
<td>0x0104 [SRAM]</td>
</tr>
<tr>
<td>g_inputCourse</td>
<td>0x0105</td>
<td>char*</td>
<td>0x010A [SRAM]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>67 'C'</td>
<td>char</td>
<td>0x0105 [SRAM]</td>
</tr>
<tr>
<td>g_a</td>
<td>0 ''</td>
<td>char</td>
<td>0x0120 [SRAM]</td>
</tr>
<tr>
<td>g_b</td>
<td>0 ''</td>
<td>char</td>
<td>0x011F [SRAM]</td>
</tr>
<tr>
<td>course</td>
<td>[... ]</td>
<td>const char[9]</td>
<td>0x1100 [SRAM]</td>
</tr>
<tr>
<td>[0x00]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1100 [SRAM]</td>
</tr>
<tr>
<td>[0x01]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1101 [SRAM]</td>
</tr>
<tr>
<td>[0x02]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1102 [SRAM]</td>
</tr>
<tr>
<td>[0x03]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1103 [SRAM]</td>
</tr>
<tr>
<td>[0x04]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1104 [SRAM]</td>
</tr>
<tr>
<td>[0x05]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1105 [SRAM]</td>
</tr>
<tr>
<td>[0x06]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1106 [SRAM]</td>
</tr>
<tr>
<td>[0x07]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1107 [SRAM]</td>
</tr>
<tr>
<td>[0x08]</td>
<td>-1 'ÿ'</td>
<td>const char</td>
<td>0x1108 [SRAM]</td>
</tr>
<tr>
<td>inputCourse</td>
<td>0xFFFF</td>
<td>char*</td>
<td>0x1109 [SRAM]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-1 'ÿ'</td>
<td>char</td>
<td>0xFFFF [SRAM]</td>
</tr>
<tr>
<td>a</td>
<td>-1 'ÿ'</td>
<td>char</td>
<td>0x110B [SRAM]</td>
</tr>
<tr>
<td>b</td>
<td>0 ''</td>
<td>char</td>
<td>0x011E [SRAM]</td>
</tr>
<tr>
<td>i</td>
<td>-1 'ÿ'</td>
<td>char</td>
<td>0x110C [SRAM]</td>
</tr>
</tbody>
</table>
## Memory mapping after execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_course</td>
<td>[...]</td>
<td>const char[5]</td>
<td>0x0100 [SRAM]</td>
</tr>
<tr>
<td>[0x00]</td>
<td>67 'C'</td>
<td>const char</td>
<td>0x0100 [SRAM]</td>
</tr>
<tr>
<td>[0x01]</td>
<td>79 'O'</td>
<td>const char</td>
<td>0x0101 [SRAM]</td>
</tr>
<tr>
<td>[0x02]</td>
<td>77 'M'</td>
<td>const char</td>
<td>0x0102 [SRAM]</td>
</tr>
<tr>
<td>[0x03]</td>
<td>80 'P'</td>
<td>const char</td>
<td>0x0103 [SRAM]</td>
</tr>
<tr>
<td>[0x04]</td>
<td>0 ''</td>
<td>const char</td>
<td>0x0104 [SRAM]</td>
</tr>
<tr>
<td>g_inputCourse</td>
<td></td>
<td>char*</td>
<td>0x010A [SRAM]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>67 'C'</td>
<td>char</td>
<td>0x0105 [SRAM]</td>
</tr>
<tr>
<td>g_a</td>
<td>0 ''</td>
<td>char</td>
<td>0x0120 [SRAM]</td>
</tr>
<tr>
<td>g_b</td>
<td>0 ''</td>
<td>char</td>
<td>0x011F [SRAM]</td>
</tr>
<tr>
<td>course</td>
<td>[...]</td>
<td>const char[9]</td>
<td>0x10F2 [SRAM]</td>
</tr>
<tr>
<td>[0x00]</td>
<td>67 'C'</td>
<td>const char</td>
<td>0x10F2 [SRAM]</td>
</tr>
<tr>
<td>[0x01]</td>
<td>79 'O'</td>
<td>const char</td>
<td>0x10F3 [SRAM]</td>
</tr>
<tr>
<td>[0x02]</td>
<td>77 'M'</td>
<td>const char</td>
<td>0x10F4 [SRAM]</td>
</tr>
<tr>
<td>[0x03]</td>
<td>80 'P'</td>
<td>const char</td>
<td>0x10F5 [SRAM]</td>
</tr>
<tr>
<td>[0x04]</td>
<td>57 '9'</td>
<td>const char</td>
<td>0x10F6 [SRAM]</td>
</tr>
<tr>
<td>[0x05]</td>
<td>48 '0'</td>
<td>const char</td>
<td>0x10F7 [SRAM]</td>
</tr>
<tr>
<td>[0x06]</td>
<td>51 '3'</td>
<td>const char</td>
<td>0x10F8 [SRAM]</td>
</tr>
<tr>
<td>[0x07]</td>
<td>50 '2'</td>
<td>const char</td>
<td>0x10F9 [SRAM]</td>
</tr>
<tr>
<td>[0x08]</td>
<td>0 ''</td>
<td>const char</td>
<td>0x10FA [SRAM]</td>
</tr>
<tr>
<td>inputCourse</td>
<td>0x0115</td>
<td>char*</td>
<td>0x10FB [SRAM]</td>
</tr>
<tr>
<td>-&gt;</td>
<td>67 'C'</td>
<td>char</td>
<td>0x0115 [SRAM]</td>
</tr>
<tr>
<td>a</td>
<td>-1 'ý'</td>
<td>char</td>
<td>0x10FD [SRAM]</td>
</tr>
<tr>
<td>b</td>
<td>0 ''</td>
<td>char</td>
<td>0x011E [SRAM]</td>
</tr>
<tr>
<td>i</td>
<td>10 ' '</td>
<td>char</td>
<td>0x10FE [SRAM]</td>
</tr>
</tbody>
</table>
Memory mapping diagram

Static data

0x0100: g_course
0x0104: Constants
0x0105: g_inputCourse
0x0109: pointer (g_inputCourse)
0x0115: inputCourse
0x011D: b
0x011E: g_b
0x011F: g_a

Dynamic data

0x10F2: course constants
0x10FA: pointer (inputCourse)
0x10FAB: a
0x10FAC: g
0x10FD: i
0x10FE: RAMEND
0x10F2: isCOMP9032
Remarks

- Data has scope and duration in the program
- Data has types and structure
- 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 types.
  - Implementation of advanced data structures/variables will be covered later.
Example 1

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

```c
int a;
unsigned int b;
char c;
char* d;
```
Example 1: solution

- Translate the following variables. Assume each:
  - `dseg` ; in data memory
  - `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 SRAM
- Labels are given the same name as the variable for convenience.
Example 2

- Translate the following C constants and variables.

C code:

```c
int a;
const char b[] = "COMP2121";
const int c = 2121;
```

Assembly code:

```assembly
.dseg
a: .byte 4
.cseg
b: .db "COMP2121", '\0'
C: .dw 2121
```

- All variables are in SRAM and constants are in FLASH
Example 2 (cont.)

- An insight of the memory mapping
  - In program memory, data is packed into words. If only a single byte is left, that byte is stored in the high byte and the low byte is filled with 0.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>'C'</td>
<td>'O'</td>
</tr>
<tr>
<td>0x0001</td>
<td>'M'</td>
<td>'P'</td>
</tr>
<tr>
<td>0x0002</td>
<td>'2'</td>
<td>'1'</td>
</tr>
<tr>
<td>0x0003</td>
<td>'2'</td>
<td>'1'</td>
</tr>
<tr>
<td>0x0004</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x0005</td>
<td></td>
<td>2121</td>
</tr>
</tbody>
</table>

Hex values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>4F</td>
</tr>
<tr>
<td>4D</td>
<td>50</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>49</td>
<td>08</td>
</tr>
</tbody>
</table>
Example 3

- Translate data structures

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

typedef struct STUDENT_RECORD *student;

student s1;
student s2;
```
Example 3: solution

- Translate data structures

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

.dseg
s1:   .byte  STUDENT_RECORD_SIZE
s2:   .byte  STUDENT_RECORD_SIZE
```
Example 4

- Translate data structures
- with initialization

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

typedef struct STUDENT_RECORD *student;

student s1 = {123456, "John Smith", 75};
student s2;
```
Example 4: solution

- Translate data structures

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

cseg
s1_value: .dw HWRD(123456)
          .dw LWRD(123456)
          .db "John Smith",0,0,0,0,0,0,0,0,0,0,0
          .db 75

.dseg
s1:   .byte STUDENT_RECORD_SIZE
s2:   .byte STUDENT_RECORD_SIZE
```
Remarks

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

• In the assembly program, you can use expressions for values.
• When 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 are internally 32 bits.

- Example

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

Function    Operands   Operator
Operands

- Operands can be
  - 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): 10, 255
      - Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
      - Binary: 0b00001010, 0b11111111
      - Octal (leading zero): 010, 077
  - PC
    - Program counter value.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>&lt;&lt;</td>
<td>Shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</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>&amp;</td>
<td>Bitwise And</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise Xor</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical And</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Same meanings as in C
Functions

- **LOW(expression)**
  - Returns the low byte of an expression
- **HIGH(expression)**
  - Returns the second 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 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)
Example 1

; Example1:

ldi r17, 1<<5 ; load r17 with 1
; shifted left 5 times
Example 2

;Example 2: compare r17:r16 with 3167

    cpi r16, low(3167)
    ldi r18, high(3167)
    cpc r17, r18
    brlt case1

... case1: inc r10
Macros

- A sequence of instructions in an assembly program often needs to be repeated several times
- Macros help programmers to write code efficiently and nicely
  - Type/define a section code once and reuse it
    - Neat representation
  - Like an inline function in C
    - When assembled, the macro definition is expanded at the place it was used.
Detectives for Macros

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

- `.endmacro`
  - Defines the end of a macro definition.
Macros (cont.)

● Macro definition structure:

```
.macro name
  ; macro body
.endmacro
```

● Use of Macro

```
macro_name [param0, param1, ..., param9]
```
Example 1

- Swapping memory data p, q twice

**With macro**

```assembly
.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
swap1
```

**Without macro**

```assembly
lds r2, p
lds r3, q
sts q, r2
sts p, r3
lds r2, p
lds r3, q
sts q, r2
sts p, r3
```
Example 2

Swapping any two memory data

```assembly
.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

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

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

end: rjmp end
```
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, we access program memory to load data
Load Program Memory

- **Syntax:** `lpm Rd, Z+`
- **Operands:** Rd \(\in\) \{r0, r1, ..., r31\}
- **Operation:** Rd \(\leftarrow\) (Z)
- \[ Z \leftarrow Z + 1 \text{ (optional)} \]
- **Words:** 1
- **Cycles:** 3
Load from program memory

- The address label in the memory program is word address
  - Used by the PC register
- To access data, the byte address is used.
- Address register, Z, is used to point bytes in the program memory
Example

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

ldi ZH, high(Table_1<<1) ; Initialize Z-pointer
ldi ZL, low(Table_1<<1)

lpm r16, Z ; Load constant from Program
            ; memory pointed to by Z (r31:r30)

Table_1:
    .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

```c
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

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

.cseg rjmp start ; jump over data definitions

s1_value: .dw HWRD(123456)
 .dw LWRD(123456)
 .db "John Smith",0,0,0,0,0,0,0,0,0,0
 .db 75

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

.dseg
s1: .byte STUDENT_RECORD_SIZE
```
Complete example 2

- Convert lower-case to upper-case for a string
  - The string is stored in the program memory
  - The resulting string after conversion is stored in data memory.
  - In ASCII, upper case letter + 32 = low case letter
**Complete example 2 (cont.)**

**Assembly program**

```
.include "m2560def.inc"
.equ size = 5
.dseg ; Set the starting address
Cap_string: .byte 5
.cseg
    rjmp start ; Skip over data
Low_string: .db "hello",0
start: ldi ZL, low(Low_string<<1) ; Get the low byte of
    ; the address of "h"
    ldi ZH, high(Low_string<<1) ; Get the high byte of
    ; the address of "h"
    ldi YH, high(Cap_string)
    ldi YL, low(Cap_string)
    clr r17 ; counter=0
```
### Complete example 2 (cont.)

#### Assembly program (cont.)

```assembly
main:
  lpm r20, Z+ ; Load a letter from flash memory
  subi r20, 32 ; Convert it to the capital letter
  st Y+,r20    ; Store the capital letter in SRAM
  inc r17      ; counter++
  cpi r17, size ; counter < size
  brlt main

loop:
  rjmp loop
```
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 steps in the assembly processes:
  - Pass one
  - Pass two
Two Passes in Assembly

- Pass one
  - Lexical and syntax analysis: checking for syntax errors
  - Record all the symbols (labels etc) in a symbol table
  - Expand macro calls

- 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**

```assembly
.equ bound = 5

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

end:
  rjmp end
```

**Symbol table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>5</td>
</tr>
<tr>
<td>loop</td>
<td>1</td>
</tr>
<tr>
<td>end</td>
<td>5</td>
</tr>
</tbody>
</table>
Example (cont.)

Code generation

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Assembly statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000:</td>
<td>2700</td>
<td>clr r16</td>
</tr>
<tr>
<td>000000001:</td>
<td>3005</td>
<td>cpi r16,0x05</td>
</tr>
<tr>
<td>000000002:</td>
<td>F010</td>
<td>brlo PC+0x02</td>
</tr>
<tr>
<td>000000003:</td>
<td>9503</td>
<td>inc r16</td>
</tr>
<tr>
<td>000000004:</td>
<td>CFFC</td>
<td>rjmp PC-0x0004</td>
</tr>
<tr>
<td>000000005:</td>
<td>CFFF</td>
<td>rjmp PC-0x0001</td>
</tr>
</tbody>
</table>
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

1. Source file with location information (NAME.ASM)
2. Absolute assembler
3. Executable file (NAME.EXE)
4. Loader Program
5. Computer memory
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

Source file 1 (MODULE1.ASM)

Relocatable assembler

Object file 1 (MODULE1.OBJ)

Source file 2 (MODULE1.ASM)

Relocatable assembler

Object file 2 (MODULE2.OBJ)

Library of object files (FILE.LIB)

Code and data location information

Linker program

Executable file (NAME.EXE)
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.

3. Discuss the advantages of using Macros. Do macros help programmer write an efficient code? Why?
Homework

4. Write an assembly program to find the length of a string. The string is stored in the program memory and the length will be stored in the data memory.
5. Write an assembly program to find the student average WAM in a class. The record for each student is defined as

```assembly
struct STUDENT_RECORD
{
    int student_ID;
    char name[20];
    char WAM;
};
typedef struct STUDENT_RECORD *student;
```

Assume there are 5 students and all records are stored in the program memory. The average WAM will be stored in the data memory.
Reading Material

- Chap. 5. Microcontrollers and Microcomputers
- 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.