# OBJECTIVES

- Become introduced to AVR® assembly programming and the *ATxmega128A1U* microcontroller.
- Further understand how to utilize *Microchip/Atmel Studio* for creating, simulating, and emulating a program.
- Design an assembly program to filter and store data based on certain criteria.

# INTRODUCTION

The foundation of every computer architecture is a specific set of operations. In general, each operation within a computer architecture can be referenced by a unique numeric value known as an **op**eration **code** (**opcode**)[1], and the set of all operation codes for an architecture defines a low-level programming language often referred to as **machine code**. Additionally, an operation code is also generally given a symbolic name, classifying it as an **instruction**. The collection of all instructions for a given architecture then defines the most abstract level of the architecture, often referred to as the **I**nstruction **S**et **A**rchitecture (**ISA**), or more frequently, the **assembly language**.

With an assembly language, a **computer program**[2] can be written much more quickly than with a machine code. However, for a given computer architecture to be able execute a program written in an assembly language, the program must first be converted into an appropriate machine code format with a pre-built software program often referred to as an **assembler**[3], and then stored into some appropriate computer memory.

# LAB STRUCTURE

In this lab, you will start to gain familiarity with the *ATxmega128A1U* microcontroller (generally referred to in this course as the *XMEGA*) as well further understand how to leverage *Microchip/Atmel Studio* (referred to in the rest of this document as *Atmel Studio*). First, you will learn various fundamental information regarding the *ATxmega128A1U*, the AVR assembler, and *Atmel Studio*. Then, you will begin to utilize the AVR ISA to design your first AVR assembly language program. After creating this program, your microcontroller will be able to filter and store data based on several given conditions.

---

RA

# REQUIRED MATERIALS

- Atmel XMEGA AU Manual (doc8331)
- Atmel *ATxmega128A1U* Manual (doc8385)
- AVR Instruction Set (doc0856)
- AVR Assembler User Guide *(includes assembler directives)*
- *Getting Started with Atmel Studio 7 (User Guide)*
- *OOTB µPAD v2.0* with USB A/B cable
- **D**igilent **A**nalog **D**iscovery (**DAD**) with *Waveforms* software

# SUPPLEMENTAL MATERIALS

- Assembly Language Conversion: GCPU to AVR
- Utilizing *Watch* in *Atmel Studio*
- Assembly Auto Complete Extension User Guide
- `lab1_f24_skeleton.asm`

---

[1] Operation codes are often represented in terms of fields of zeroes and ones, otherwise referred to as **bit fields**, where these fields represent a unique, encoded binary number. For hardware to handle bit fields, some form of encoding/decoding circuitry must be utilized.

[2] A **program**, or **program of execution**, is a collection of either computer instructions or operation codes that direct which operations a computer is to perform.

[3] Normally, assemblers are paired with an additional software program, generally known as a **preprocessor**, to recognize additional keywords, known as **assembler directives**, which specify actions to be performed directly by the assembler (and not by the computer for which the code is assembled).

# PRE-LAB PROCEDURE

## REMINDER OF LAB POLICY

You must re-read the *Lab Rules and Policies* before submitting any pre-lab assignment and before attending any lab.

Throughout this course, it will be necessary to conduct individual research. When doing so, various types of documentation, e.g., manuals, datasheets, application notes, and tutorials, will all be of interest.

Below, you will begin to become exposed to some important documentation relevant to this course, and will learn some basics regarding the *ATxmega128A1U* microcontroller, the AVR assembler, and *Atmel Studio*.

1. Study §§ 1-4 of the Atmel XMEGA AU Manual (doc8331), a more general user manual describing the XMEGA AU computer architecture. Additionally, study §§ 3, 6, and 7 of the Atmel *ATxmega128A1U* Manual (doc8385), the specific datasheet for the ATxmega128A1U microcontroller. Then, skim through the AVR Instruction Set (doc0856) to get an idea of the operations available to AVR microcontrollers. Next, read AVR Assembler User Guide, especially section 5 on *Assembler Directives*. Finally, look through the following the *Getting Started with Atmel Studio 7 (User Guide)*, especially the part on debugging (§§ 1.13-1.15).

## PRE-LAB EXERCISES

i. As specified in Lab 0, you should have, upon receiving your kit, verified that it contained all of the parts listed on *µPAD v2.0 Parts List* (*Excel* or *PDF*). If it did not, you should have immediately notified the PI if any components were missing. For documentation purposes (and before any assembly), you should have taken pictures of all of the parts in your kit (each of the PCBs, the chips, etc.). Include these images in your **Lab 1** *Pre-Lab Report*. Your **Lab 1** report should also include images of all of your now **completely constructed** PCBs.

ii. Which type of memory alignment is used for program memory in the *ATxmega128A1U*? Byte-alignment, or word-alignment? What about for data memory?

iii. Which assembly instructions can be used to load data *indirectly* from data memory within XMEGA AU microcontrollers? Which assembly instructions can be used to store data *indirectly* to data memory?

iv. Which assembly instruction can be used to load data *directly* from **any** of the general purpose I/O memory of XMEGA AU microcontrollers? Which assembly instruction can be used to store data *directly* to **any** of the I/O memory?

v. Which assembler directive places a byte of data in program memory? Which assembler directive allocates space within data memory? Which assembler directives allow you to provide expressions (either constant or variable) with a meaningful name?

vi. Which assembly instructions can be used to read from (flash) program memory? For each instruction, list which registers can be used as an operand.

vii. Your Lab 1 report should also include images of all of your complete constructed PCBs.

viii. In which section of program memory is address 0xF086 located?

ix. If you were to use the *Memory* debug window of *Atmel Studio* to verify that some datum was correctly stored at address 0x9B4F within program memory of the *ATxmega128A1U*, which address would you specify within the debug window?

x. When using the internal SRAM (not EEPROM), which memory locations can be utilized for the data segment (`.dseg`)? Why?

xi. Which is the first (i.e., lowest) *program memory* address (this is an address to the 16-bit wide program memory information) that would require the relevant RAMP register to be changed from its initial value of zero? Why?

xii. In the context of pointing an index to a specific program memory address within an XMEGA AU architecture, explain why and how the address value should first be altered. Similarly, in the context of pointing an index to a specific data memory address, explain why the address value should not be altered.

Now, you will design your first AVR® assembly language program, **`lab1.asm`**. A skeleton for this file is provided on our course website and is also available through the *Supplemental Materials* section of this document.

Overall, this program should filter data stored within a predefined input table based on a set of given conditions and store a subset of filtered values into an output table. More specifically, the following bulleted list describes an algorithm that should be performed on each item within the predefined input table, until an **e**nd-**o**f-**t**able (**EOT**) value of *NULL*, defined to be zero, is encountered within the table. Note that the algorithm should be executed in the same order as the bulleted

list is provided, and for each iteration of the algorithm, only one of the three overall conditions within the bulleted list should be performed.

- Upon finding the EOT value within the input table, the output table should be terminated with a NULL character.
- If bit 6 is set, divide the 8-bit value by 2 (unsigned); if that result is greater than or equal to 95, add 2 and store it to the next available location within the output table.
- Else, multiply by 2 and check if the product is less than 77; if it is, then subtract 8 from it and store the result to the next available location within the output table.

The following are additional specifications for the program:

- The input table should be placed in program memory, starting at address 0xF086, and should consist of the 8-bit data provided in the left-hand of Table 1, where each value in this column should be stored sequentially in memory, without any padding, in the same format provided. (Data is given in decimal, hexadecimal, binary, octal, and ASCII formats to demonstrate that *Atmel Studio* can interpret values in each of these given formats.)
- An output table should be allocated within data memory, starting at address 0x2783.
- All values should be interpreted as unsigned, i.e., instructions that interpret data as a signed value (e.g., BRLT, BRGE, etc.) should not be utilized.

If the program is written as specified above, the resulting output table should contain the message "*ButterDog*" when the relevant data is viewed in terms of the ASCII encoding format supported by *Atmel Studio*. To view the data in this format, a *Memory* debug window within *Atmel Studio* should be utilized.

**NOTES:**

- The second column of Table 1 provides the relevant input table data in terms of the ASCII encoding format supported by *Atmel Studio*, simply to allow ease of verification if debugging with a *Memory* view window in *Atmel Studio*. (In general, ASCII provides a standard set of encoded values, often in terms of seven bits although sometimes in terms of eight bits if additional symbols are supported, for commonly used symbols in human language. An example ASCII table can be found at http://www.asciitable.com/.)
- In order to make your code modular and portable (i.e., able to be reused in different contexts), utilize assembler directives. For example, use assembler directives to create constant or variable identifiers for pertinent memory addresses within your input/output tables, for EOT values, etc.
- A *Watch* window, available under *Debug* | *Windows* | *Watch* within *Atmel Studio*, is used to view memory locations while debugging a program; to learn more about *Watch* windows, navigate to *Debugging* | *Memory View* within the *Getting Started with Atmel Studio 7 (User Guide)*, as well as to the Using *Watch* in *Atmel Studio* document located on the course website, listed under *Software/Docs*.
- To facilitate programming in the AVR assembly language within *Atmel Studio*, it is recommended that you install the *Auto Complete Extension* created by a former *4744* student. To learn how to do so, refer to the *Supplemental Materials* section of this document.

**Table 1:** Memory Table

| Data | Data (ASCII)[1] |
|---|---|
| 37 | % |
| 127 | Not Visible (Delete) |
| 'æ' | æ |
| 0xE4 | ä |
| '?' | ? |
| 0b11100100 | ä |
| 'j' | j |
| 0b11000110 | Æ |
| 224 | à |
| 0x37 | 7 |
| 38 | & |
| 0b01111101 | } |
| 'Ú' | Ú |
| 202 | Ê |
| 0x00 | Not Visible (Null) |

[1]ASCII characters are standard for values less than 128 = 0x80. Characters at 128 and above are not unique and many are not visible easily represented in a table. *Microchip/Atmel Studio* extended ASCII characters at 128 and above are shown.

1. Make a flowchart or write pseudocode for the above program. (This is required for **ALL** lab programs and may not be specifically requested in future lab documents.)
2. Create the relevant assembly language program, **`lab1.asm`**, as specified above. A skeleton for this file is provided on our course website and is also available through the *Supplemental Materials* section of this document.
3. Test your program using the *Atmel Studio* software simulator. Utilize debugging tools to verify that the program works as specified.
4. Emulate the program on your *μPAD* to verify that the program also works on your hardware. Utilize the same debugging tools.
5. Take a screenshot of a *Memory* view window after executing the relevant program, showing the entire output table at the appropriate memory locations.

## PRE-LAB PROCEDURE SUMMARY

1) Read the specified sections within the relevant documentation.
2) Solve all pre-lab exercises.
3) Make a flowchart or write pseudocode for the described program.
4) Write the relevant assembly program, **`lab1.asm`**. Verify its correctness.
5) Capture a screenshot of a *Memory* view window displaying the resulting output table after executing the program.