# ECE 375: Computer Organization and Assembly Language Programming

Lab 4 – Large Number Arithmetic

## SECTION OVERVIEW

**Complete the following objectives:**

- Understand and use arithmetic/ALU instructions.
- Manipulate and handle large (> 8 bits) numbers.
- Create and handle functions and subroutines.
- Verify the correctness of large number arithmetic functions via simulation.

***Note, this lab requires to use Windows computers.***

## PRELAB

### Introduction

Writing assembly code requires a lot of attention to detail. Despite your best intentions, it is easy to introduce subtle bugs into your program, especially when you are first learning assembly language programming. Often, you will try to run your code on your `mega32U4` board, observe that it isn't working, but have a difficult time identifying which portion of your program isn't working as intended. In other words, **simple mistakes** can result in behavior that is **significantly different** than what you intended, which is often confusing or frustrating. Therefore, it is very useful to know how to to simulate your code, so that you can provide "controlled" input and observe that each portion of your code is working as you intended.

In this lab, you will learn to use the simulator to step through a sample program, observing and recording key values along the way. However, due to a compatibility issue with the simulator, **we will use `Atmega128` architecture only for this lab**. You may already see the pre-compiler directive file (`m128def.inc`) for `Atmega128` is included in the skeleton code. Please make sure to select `ATmega128` when creating a new project. Plus, be sure to write down your answers to all **bolded questions** as you proceed through the lab, as you will need them at the end of the lab.

### Creating Breakpoints

To get started, complete the following steps to establish a new project with the sample program included:

1. Open Atmel Studio, and create a new Assembler project.
2. Download the sample assembly program given on the lab webpage (`PrelabSample.asm`), and include it into your project.
3. Make sure that the sample program builds without any errors.

In the sample program you just downloaded, there are several lines that have the comment "SET BREAKPOINT HERE". Before you simulate this sample code, you will need to add a breakpoint to every one of these lines. Do the following for each line that has a "SET BREAKPOINT HERE" comment:

- Right-click on the line of code, and then select **Breakpoint** → **Insert Breakpoint**.
- If done correctly, a red dot will appear to the left of the instruction, and the entire line will be highlighted in red.

For this lab to work correctly, it is very important that all required breakpoints are set. Once you have completed this step, and verified that all required breakpoints are in place, you are ready to begin simulating the sample code.

### Preparing for Simulation

To start up the simulator, select **Debug** → **Start Debugging and Break** from the Atmel Studio menu bar (or just press Alt+F5).

Since this is your first time launching the simulator for this particular Atmel Studio project, you will be prompted to configure your `.asmproj` file with the message "Please select a connected tool and interface and try again." Press **Continue**, and then open the dropdown menu below the text "Selected debugger/programmer", and click on **Simulator**.

Then, you can press **Ctrl+S** to save this change to the project file, and then switch back over to the code editor tab, which should still have the sample program open. Finally, you can select **Debug** → **Start Debugging and Break** again and the simulator will properly start.

Once the simulator has started, you should see several new view tabs in the right-side window of Atmel Studio.

- If you do not see "Processor Status" anywhere, go to **Debug** → **Windows** → **Processor Status** to bring it up. It should auto-dock in the right-side window, if not you can drag the window over to dock it manually.
- If you do not see "I/O" anywhere, go to **Debug** → **Windows** → **I/O** to bring it up. Again, it should dock automatically, but you can dock it manually if needed.
- If you do not see any "Memory" windows open, go to **Debug** → **Windows** → **Memory** → **Memory 1** to bring one up. This memory view is traditionally docked in the lower-right panel of Atmel Studio.

Before you continue, **have your TA verify** that you have set all required breakpoints, and that you have opened the Processor Status, I/O, and Memory windows correctly.

## Simulating the Sample Code – Part 1

1. We began simulation by pressing "Start Debugging and Break", so the simulator is currently in debugging (line-by-line) mode, paused at the very first instruction of our program (it should be `rjmp INIT`). This line is highlighted in yellow, which means it is the **next** instruction to be executed.
2. Since the simulator hasn't actually run any lines of code yet, take this opportunity to observe the default/initial values of some of the I/O registers you have already seen in the previous lab.
   - **What is the initial value of DDRB?**
   - **What is the initial value of PORTB?**
   - **Based on the initial values of DDRB and PORTB, what is Port B's default I/O configuration?**
3. Press **Debug** → **Step Into** (F11) to execute the first instruction and step forward to the next line. Observe that the flow of the program has moved to the line directly following the `INIT` label.
4. Next, press **Debug** → **Continue** (F5) to enter run mode. The simulator will run until it encounters the next breakpoint.
5. At this point (Breakpoint #1), the simulator has just finished executing the 4 lines of code that initialize the stack pointer. The stack pointer's value is displayed (in hexadecimal representation) in the Processor Status window.
   - **What 16-bit address was the stack pointer just initialized to?**
6. Press **Continue** again. The program has now advanced another two lines (to Breakpoint #2), and the general purpose register `r0` has just been initialized to a certain value. The contents of general purpose registers like `r0` are also displayed in the Processor Status window.
   - **What are the current contents of register `r0`?**
7. Press **Continue** again. The program flow has continued for several lines, and then paused again at the end of the first iteration of a loop structure (Breakpoint #3), which began at the `LOOP` label. The next instruction to be executed will test whether to move the program flow back up to `LOOP` (i.e., it tests whether the loop will continue). Press **Continue** again. The simulator is still pointing to the same instruction, which means that another iteration of the loop was run. Keep pressing **Continue** until the simulator stops at the first breakpoint outside of the loop (Breakpoint #4).
   - **How many times did the code inside the loop structure end up running?**
   - **Which instruction would you modify if you wanted to change the number of times that the loop runs?**
   - **What are the current contents of register `r1`?**
8. Press **Continue** again. The program flow is now within another loop structure (Breakpoint #5). Keep pressing **Continue** until the program stops at the first breakpoint outside of the `LOOP2` loop (Breakpoint #6).
   - **What are the current contents of register `r2`?**
9. Press **Continue** to advance to the final breakpoint (Breakpoint #7).
   - **What are the current contents of register `r3`?**

## Inserting Values into Memory

Before you advance the simulator beyond Breakpoint #7, you will need to manually insert some values into the Data Memory. Go to the Memory window, and select **data IRAM** from the "Memory:" dropdown. You should now see a table of two-digit hexadecimal values (bytes), aligned so that the first entry (top left) is the contents of the first location in Data Memory (at address $0100). The entry to its right is the contents of location $0101, then to its right is $0102, etc.

The Memory window allows you to type values directly into memory, as long as the simulation is currently paused in line-by-line mode. Before stepping through any more of the sample code (i.e., before beginning the `FUNCTION` subroutine), complete the following steps:

1. Enter the value you observed in r0 into location $0100.
2. Enter the value you observed in r1 into location $0101.
3. Enter the value you observed in r2 into location $0102.
4. Enter the value you observed in r3 into location $0103.

### Simulating the Sample Code – Part 2

Once you have placed the correct values into the correct data memory locations, you can resume stepping through the sample code.

1. Press **Step Into** (F11) to move into the `FUNCTION` subroutine.
    - **What is the value of the stack pointer now that your program flow has moved inside of a subroutine?**
2. Press **Step Out** (Shift+F11), which will run the rest of the subroutine all at once and pause at the first instruction after the function call (which happens to be the end of `MAIN`).
    - **What is the final result of `FUNCTION`? (What are the hexadecimal contents of memory locations $0105:$0104?)**

### Prelab Questions

In order to recieve a full credit for the prelab report, you need to answer the following questions.

1. What is the initial value of DDRB?
2. What is the initial value of PORTB?
3. Based on the initial values of DDRB and PORTB, what is Port B's default I/O configuration?
4. What 16-bit address (in hexadecimal) is the stack pointer initialized to?
5. What are the contents of register r0 after it is initialized?
6. How many times did the code inside of `LOOP` end up running?
7. Which instruction would you modify if you wanted to change the number of times that the loop runs?
8. What are the contents of register r1 after it is initialized?
9. What are the contents of register r2 after it is initialized?
10. What are the contents of register r3 after it is initialized?
11. What is the value of the stack pointer when the program execution is inside the `FUNCTION` subroutine?
12. What is the final result of `FUNCTION`? (What are the hexadecimal contents of memory locations $0105:$0104)?
13. Write pseudocode for an 8-bit AVR function that will take two 16-bit numbers (from data memory addresses $0111:$0110 and $0121:$0120), **add them together**, and then store the 16-bit result (in data memory addresses $0101:$0100). (Note: The syntax "$0111:$0110" is meant to specify that the function will expect *little-endian* data, where the highest byte of a multi-byte value is stored in the highest address of its range of addresses.)
14. Write pseudocode for an 8-bit AVR function that will take the 16-bit number in $0111:$0110, **subtract it from** the 16-bit number in $0121:$0120, and then store the 16-bit result into $0101:$0100.

## BACKGROUND

Arithmetic calculations like addition and subtraction are fundamental operations in many computer programs. Most programming languages support several different data types that can be used to perform arithmetic calculations. As an 8-bit microcontroller, the ATmega32 primarily uses 8-bit registers and has several different instructions available to perform basic arithmetic operations on 8-bit values. Some examples of instructions that add or subtract 8-bit values contained in registers are:

```
ADD R0, R1  ; R0 <- R0 + R1
ADC R0, R1  ; R0 <- R0 + R1 + C
SUB R0, R1  ; R0 <- R0 - R1
```

If we want to perform arithmetic operations on values that are too large to represent with only 8 bits, but still use the same 8-bit microcontroller for these large number operations, then we need to develop a procedure for manipulating multiple 8-bit registers to produce the correct result.

## Multi-byte Addition

This example demonstrates how 8-bit operations can be used to perform an addition of two 16-bit numbers. (The layout of this example should look familiar; it is meant to look like the way you would usually write out an addition by hand.)

```
      Possible Carry-out of R0 + R2 Addition -> 1
                                                 R1  R0
   Possible Carry-out of R1 + R3 Addition -> 1  R3  R2
                                              + -----------
                                              R4  R3  R2
```

Initially, one of the 16-bit values is located in registers R1:R0 and the other value is in R3:R2. First, we add the 8-bit values in R0 and R2 together, and save the result in R2. Next, we add the contents of R1 and R3 together, account for a possible carry-out bit from the previous operation, and then save this result in R3. Finally, if the result of the second addition generates a carry-out bit, then that bit is stored into a third result register: R4.

Why is an entire third result register necessary? Even though our 16-bit addition can result in **at most** a 17-bit result, the ATmega32's registers and data memory words have an intrinsic size of 8 bits. As a consequence, we must handle our 17-bit result as if it's a 24-bit result, even though the most significant byte only has a value of either 1 or 0 (depending on if there was a carry-out).

## Multi-byte Subtraction

Subtracting one 16-bit number from another 16-bit number is similar to the 16-bit addition. First, subtract the low byte of the second number from the low byte of the first number. Then, subtract the high byte of the second number from the high byte of the first number, accounting for a possible borrow that may have occurred during the low byte subtraction.

When performing 16-bit *signed* subtraction, the result sometimes requires a 17th bit in order to get the result's sign correct. For this lab, we will just deal with the simpler *unsigned* subtraction, and we will also assume that the subtraction result will be positive (i.e., the first operand's magnitude will be greater than the second operand's magnitude). **Therefore, our 16-bit subtraction result can be contained in just two result registers**, unlike the 16-bit addition.

## Multiplication

The AVR 8-bit instruction set contains a special instruction for performing *unsigned* multiplication: `MUL`. This instruction multiplies two 8-bit registers, and stores the (up to) 16-bit result in registers R1:R0. This instruction is a fast and efficient way to multiply two 8-bit numbers. Unfortunately, multiplying numbers larger than 8 bits wide isn't as simple as just using the `MUL` instruction.

The easiest way to understand how to multiply large binary numbers is to visualize the "pencil & paper method", which you were likely taught when you first learned how to multiply multi-digit decimal numbers. This method is also known as the *sum-of-products* technique. The following diagram illustrates using this typical method of multiplication for decimal numbers:

```
     24
*    76
------
     24    (4 * 6 = 24)
    12_    (2 * 6 = 12, but aligned with the tens' place)
    28_    (4 * 7 = 28, but aligned with the tens' place)
+ 14__     (2 * 7 = 14, but aligned with the hundreds' place)
------
   1824
```

This method multiplies single decimal digits by single decimal digits, and then sums all of the partial products to get the final result. This same technique can be used for multiplying large binary numbers. Since there is an instruction that implements an 8-bit multiplication, `MUL`, we can partition our large numbers into 8-bit (1 byte) portions, perform a series of one byte by one byte multiplications, and sum the partial products as we did before. The diagram below shows this *sum-of-products* method used to multiply two 16-bit numbers (`A2:A1` and `B2:B1`):

```
           A2  A1
*          B2  B1
-----------------
          H11 L11    (A1 * B1 = H11:L11)
      H21 L21 ___    (A2 * B1 = H21:L21, but properly aligned)
      H12 L12 ___    (A1 * B2 = H12:L12, but properly aligned)
+ H22 L22 ___ ___    (A2 * B2 = H22:L22, but properly aligned)
-----------------
   P4  P3  P2  P1
```

The first thing you should notice is that the result of multiplying two 16-bit (2 byte) numbers yields an (up to) 32-bit (4 byte) result. In general, when multiplying two binary numbers, you will need to allocate enough room for a result that can be twice the size of the operands.

`H` and `L` signify the high and low result bytes of each 8-bit multiplication, and the numbers after `H` and `L` indicate which bytes of the 16-bit operands were used for that 8-bit multiplication. For example, `L21` represents the low result byte of 16-bit partial product produced by the `A2 * B1` multiplication.

Finally, it is worth noticing that the four result bytes are described by the following expressions:

```
P1 = L11
P2 = H11 + L21 + L12
P3 = H21 + H12 + L22 + any carries from P2 additions
P4 = H22 + any carries from P3 additions
```

## PROCEDURE

**Note that you are given 2 weeks to complete this lab.**

First, you need to implement three different large number arithmetic functions: `ADD16` (a 16-bit addition), `SUB16` (a 16-bit subtraction), and `MUL24` (a 24-bit multiplication). A pre-written `MUL16` function has been provided in the skeleton file. You may use it as the reference for `MUL24`, however, **do not assume** it will be simple as if the function will run by just changing some values. **You will most liikely need to come up with whole new ideas to handle extra bits.**

The skeleton file provides operand values for each of these functions. Each of these functions should (1) read input operands from program memory (The declared operand values in program memory needs to be loaded into data memory first), (2) conduct corresponding arithmetic operations, (3) store its result back in data memory. Note that adding any different values directly into the program memory is **NOT ALLOWED**. Also, in skeleton code file, you **should keep** the lines with 'nop' instruction set, which is where TAs will check if the operands/results are loaded correctly.

After completing and testing `ADD16`, `SUB16`, and `MUL24`, you need to write a fourth function named `COMPOUND`. `COMPOUND` uses the first three functions to evaluate the expression $((G - H) + I)^2$, where $G$, $H$, and $I$ are all unsigned 16-bit numbers. The test values you must use for $G$, $H$, and $I$ are provided in the skeleton file. `COMPOUND` should be written to automatically provide the correct inputs to `ADD16`, `SUB16`, and `MUL24`, so that you can run `COMPOUND` all at once without pausing the simulator to enter values at each step. Note that the $G$, $H$, and $I$ values are different than the values you used to individually verify `ADD16`, `SUB16`, and `MUL24`.

## STUDY QUESTIONS / REPORT

A full lab write-up is required for this lab. When writing your report, be sure to include a summary that details **what you did and why, and explains any problems you may have encountered**. Your write-up and code must be submitted by the beginning of next week's lab. Remember, NO LATE WORK IS ACCEPTED.

### Study Questions

1. Although we dealt with unsigned numbers in this lab, the ATmega32 microcontroller also has some features which are important for performing signed arithmetic. What does the $V$ flag in the status register indicate? Give an example (in binary) of two 8-bit values that will cause the $V$ flag to be set when they are added together.
2. In the skeleton file for this lab, the `.BYTE` directive was used to allocate some data memory locations for `MUL16`'s input operands and result. What are some benefits of using this directive to organize your data memory, rather than just declaring some address constants using the `.EQU` directive?