ECE 375: Computer Organization and Assembly Language Programming

Lab 5 – 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.

PRELAB

To complete this prelab, you may find it useful to look at the AVR Starter Guide and the AVR Instruction Set Manual. If you consult any online sources to help answer the prelab questions, you **must** list them as references in your prelab.

1. For this lab, you will be asked to perform arithmetic operations on numbers that are **larger than 8 bits**. To be successful at this, you will need to understand and utilize many of the various arithmetic operations supported by the AVR 8-bit instruction set. List and describe **all** of the addition, subtraction, and multiplication instructions (i.e. `ADC`, `SUBI`, `FMUL`, etc.) available in AVR's 8-bit instruction set.
2. 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.)
3. 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 ATmega128 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 ATmega128'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

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 should use it as the basis for `MUL24` (or complete the challenge instead, which has you implement an alternate approach to multiplication).

Each of these functions should read its input operands from data memory, and also store its result in data memory. The skeleton file provides test values for each of these functions, which you **must** use to demonstrate that your functions are working correctly. You have to declare these test values in program memory and then have your program load them into data memory. Entering test values directly into the simulators's memory is not allowed. In general, **memory organization is left up to you in this lab**, so come up with a system that makes it easy for you to debug your functions and quickly verify that your answers are correct.

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 $((D - E) + F)^2$, where D, E, and F are all unsigned 16-bit numbers. The test values you must use for D, E, and F 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. **Please note** that the D, E, and F values are different than the values you used to individually verify `ADD16`, `SUB16`, and `MUL24`.

This is a simulation-based lab; to demonstrate that your four functions work correctly, have the TA observe the results of `ADD16`, `SUB16`, `MUL24`, and `COMPOUND` and confirm that everything is working correctly. You have to understand using memory properly. TAs will check your functions with 4 break points (`ADD16`, `SUB16`, `MUL24`, and `COMPOUND`).

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 ATmega128 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?

CHALLENGE

To receive challenge credit for this lab, you must implement a 24-bit multiplication using the following *shift-and-add* technique, instead of using the *sum-of-products* technique described in the main part of the lab handout. If you complete the challenge section, you do not need to implement the *sum-of-products* multiply, but you do still need to implement `ADD16`, `SUB16`, and `COMPOUND`.

Although the *sum-of-products* technique is easier for humans to use when performing multiplication, it is not the most efficient way of multiplying binary numbers, especially very large (e.g., 1024-bit) numbers. So, a different technique can be used, one that takes into account the inherent properties of a base-2 number system. This technique is known as *shift-and-add* multiplication.

In basic mathematics terminology, multiplication has two operands, the *multiplicand* (i.e., the operand to be multiplied) and the *multiplier*. When two decimal numbers are multiplied, the multiplication is carried out by essentially adding the multiplicand to itself over and over again a certain number of times; this number is specified by the multiplier.

This may seem fairly obvious, until you consider the analogous operation in binary. In binary, the bits of the multiplier operand are used to determine *whether* the multiplicand is added to itself or not. The example diagrams below will demonstrate both decimal and binary *shift-and-add* multiplications.

Example: Decimal *shift-and-add*

```
   23 <- Multiplicand
*  16 <- Multiplier
-----
  138    (6 * 23 = 23 + 23 + 23 + 23 + 23 + 23 = 138)
+ 230    (1 * 23 = 23, but then shift left once to get 230)
-----
  368 <- Product
```

In the above example, the decimal digit in the ones' place of the multiplier is 6, and therefore 6 instances of the multiplicand are added together to yield the value 138. The decimal digit in the tens' place of the multiplier is 1, which yields a value of 23 that then needs to be shifted left by one decimal digit to ultimately yield 230. Adding the two partial products together results in our product, 368. The following expression explains why this works:

$$23*16 = (23*6) + (23*10)$$

Example: Binary *shift-and-add*

```
Binary(4-bit):    1011 <- Multiplicand
                * 1101 <- Multiplier
Multiplier   ---------
  (LSB) 1    0000 1011 <- 1 * 1011 = 1011, then shift left 0 times
        0  + 0000 0000 <- 0 * 1011 = 0000, then shift left 1 time
             ---------
             0000 1011 <- Add results together
        1  + 0010 1100 <- 1 * 1011 = 1011, then shift left 2 times
             ---------
             0011 0111 <- Add results together
  (MSB) 1  + 0101 1000 <- 1 * 1011 = 1011, then shift left 3 times
             ---------
             1000 1111 <- Add results together for final product
```

In this example, we've performed the same *shift-and-add* technique as before, but in a binary configuration that uses 4-bit values. The basic principle is that we shift through the multiply. When a 1 is received, we add the multiplicand to the low bit of the result, if a zero is received we do nothing. We then shift the entire result one bit to the left, thus essentially multiplying the result by 2 and repeat the process. The following binary expression explains why this works:

$$1011 * 1101 = (1011 * 0001) + (1011 * 0000) + (1011 * 0100) + (1011 * 1000)$$

To make things even clearer, the right-hand side of the above binary expression is equivalent to:

$$1011 * 1101 = (1011 \texttt{<<} 0) + (0000 \texttt{<<} 1) + (1011 \texttt{<<} 2) + (1011 \texttt{<<} 3)$$

Although this method is sound and easily understandable, there is a more efficient method. Assume you are running on a 4-bit system where the registers are 4 bits wide. The binary method shown above would require 4 registers. A better approach would be to combine the lower result register with the multiplier register, so that you can shift all the registers at once and only use 3 registers instead of 4. In order perform the multiplication in this more space-efficient way, you must shift everything to the right (instead of to the left), and also rotate through the carry flag. If the carry flag is set after a rotation, then add the multiplicand to the upper result register, and proceed to the next rotation; otherwise, skip the addition and simply rotate again. Keep in mind that is very important to rotate to the right, rather than just shift, so that whatever is in the carry flag will be shifted into the result MSB, and the result LSB will be shifted out into the carry flag. The following example illustrates this more efficient method:

Example: Binary *shift-and-add* (more efficient)

```
               1011 <- Multiplicand
             * 1101 <- Multiplier
Carry     ---------
          0000 1101 <- Load Multiplier into low register
1         0000 0110 <- Rotate right through carry
          1011      <- Carry is set, so add multiplicand
          ---------
0         1011 0110 <- Result of addition
0         0101 1011 <- Rotate right through carry
          ----      <- Don't add since carry is 0
          ---------
0         0101 1011 <- Result thus far
1         0010 1101 <- Rotate right through carry
          1011      <- Add multiplicand since carry is set
          ---------
0         1101 1101 <- Result of addition
1         0110 1110 <- Rotate right through carry
          1011      <- Add multiplicand since carry is set
          ---------
1         0001 1110 <- Result of addition
0         1000 1111 <- Result after final shift, note that a '1'
                       was shifted in, because it was the carry
                       that was set from the last addition
```

As you can see, this approach to the *shift-and-add* technique can be easily implemented with a simple for loop, with a specific number of iterations that depends on the size of the data. In this example, we used 4-bit numbers, so we looped 4 times. Inside the loop, there is a simple rotate right, and an addition depending on the status of the carry bit after the rotation. This loop implementation of *shift-and-add* can be easily used for binary numbers of any width with minimal effort, and is actually used internally for multiplication in most microarchitectures.