Lab 4
Large Number Arithmetic

Objectives

- Understand and use arithmetic and ALU operations
- Manipulate and handle large numbers
- Create and handle functions and subroutines

Prelab

- For this lab, you will be asked to perform arithmetic operations on numbers that are larger than 8-bits. To do this, you should understand the different arithmetic operations supported by the AVR Architecture. List and describe all the different forms of ADD, SUB, and MUL (i.e. ADC, SUBI, MULF, etc.).
- Write pseudo-code that describes a function that will take two 16-bit numbers in data memory addresses $0110-$0111 and $0121-$0121 and add them together. The function will then store the resulting 16-bit number at the address $0100-$0101. (Hint: The upper address in the high byte of the number and don't forget about the carry in bit.)

Procedure

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.

The ATmega128 uses 8-bit registers and has several different instructions to perform basic arithmetic operations on 8-bit operands. Examples of instructions that add or subtract 8-bit registers are:

```
ADD   R0, R1        ; R0 ← R0 + R1
ADC   R0, R1        ; R0 ← R0 + R1 + Carry Bit
SUB   R0, R1        ; R0 ← R0 – R1
```

If we are required to manipulate data with 16 or more bits with AVR instructions then we need to perform intermediate steps to manipulate 8-bit registers to produce the correct result. The following figure demonstrates the addition of two 16-bit numbers. One of the 16-bit numbers is located in registers R0 and R1 and the other number is in R2 and R3. The operation result is placed in the registers R4, R5, and R6.

```
                                    (1)     (1)    ← Possible Carry-In Bit
                                    R0      R1
Possible Carry-Out Bit →     (1)    R2      R3
                          + ----------------------
                             R4     R5      R6
```

As this calculation demonstrates, we need to add R1 to R3 and accommodate for any carry-in bit. The result of this operation is stored in R6. Next, we add R0 to R2 and accommodate for any carry-in bit from the previous operation. If there is a final carry-bit, then it is stored in R4. Note that since we want to reserve the carry out bit from the operation, we actually get a 17-bit result: 16-bit plus the carry. But since the AVR architecture inherently supports 8-bit numbers and we use 3 registers, we treat the result as a 24-bit number, where the most significant byte has the value of either 1 or 0 depending on the carry out bit.

The AVR instruction set contains a special instruction to perform multiplication, MUL. This instruction multiplies two registers and stores the result in registers R0 and R1. Therefore a multiplication of 2 single byte registers generates a 2-byte result. This instruction can be used as a fast and efficient way to multiply two 8-bit numbers. Unfortunately, it can be very complicated to use when multiplying 16 or more bit numbers.

The easiest way to understand how to multiply large numbers is to visualize it like using the paper method to multiply, this method is also known as the sum of products technique. The following figure illustrates the typical paper method.

```
    24
*   76
-------
 24  (4x6=24)
12-  (2x6=12, shifted by one)
   28-    (4x7=28, shifted by one)
+ 14--    (2x7=14, shifted by two)
-------
  1824
```

As you can see, the paper method multiplies small numbers (numbers less than 10) and then adds all the products to get the final result. We can use this same technique for large binary numbers. Since there is an instruction that supports multiply, MUL, we can multiply two 8-bit numbers at a time and then add up all the products. Below is an example of how to multiple two 16-bit numbers.

```
               A1  A2
      *        B1  B2
------------------
               H22 L22
           H21 L21
           H12 L12
+  H11 L11
------------------
   P1  P2  P3  P4
```

As you can see, the result of multiplying two 16-bit (or 2 byte) numbers gives a 32-bit (or 4 byte) result. In general, when multiplying to binary numbers, the result will be twice the size of the operands. For reference, the H and the L signify the high and low byte of the result of each multiplication and numbers signify which values where multiplied, B and A. For example, L21 is the resulting low byte of multiplying B2 and A1 together. Also note that the four results are just the following:

```
P1 <= H11 + carry from P2
P2 <= H21 + H12 + L11 + carry from P3
P3 <= H22 + L21 + L12
P4 <= L22
```

The skeleton code provided for the lab will contain an algorithm that multiplies 2 16-bit numbers in the fashion described above. You will need to expand on this function for the lab assignment.

Assignment

Write a program in AVR Assembly that calculates the result of (A + B)^2. A and B are 16-bit numbers or 2 bytes, thus A + B will produce a 3-byte result since we want to save the carry bit. That result will then be multiplied by itself to produce a final 6-byte result. To do this, you will need to write **two functions**, a function that adds two 16-bit numbers and produces a 24-bit result and a function that multiplies two 24-bit numbers.

The two operands will be entered in data memory during simulation time. Operand A is at the data memory location $0100 and $0101. Operand B is at data memory location $0102 and $0103. The 6-byte result will be stored in the data memory locations $0104 - $0109.

This lab will not use the TekBot platform or the AVR board. It is purely a simulation-based lab. Write and assemble the program, test it, and demonstrate the program to your TA. The TA will provide two numbers for you to run the simulation on when you submit the assignment. Turn in the write up and the source code to your TA.

Write Up

Write a short summary that details what you did and why, explain any problems you may have encountered, and answer the questions below. This write up and your code should be submitted to your TA by the beginning of class the week following the lab. NO LATE WORK IS ACCEPTED.

Additional Questions

No additional questions for this lab assignment.

Challenge

Complete the assignment with the following shift-and-add multiplication technique instead of sum of products technique described above. If you complete the challenge section, it will count for both the lab assignment and challenge portion.

Although the sum-of-products technique is easier for humans to perform multiplications, it is not the most efficient way for multiplying binary numbers, especially very large numbers like 1024-bit numbers. Also, it is very difficult to implement a sum-of-products multiplier in hardware. So a variation to the technique was created and used the inherent properties of a base-2 number system. This technique is known as the shift-and-add multiplier.

Referring back to the basic concepts of mathematics, multiplication has two operands, the multiplicand (or the operand to be multiplied) and the multiplier. When a decimal number is multiplied, it essentially adds the multiplicand to itself for the amount specified by the multiplier. This seems pretty obvious, until you use binary. In binary, the multiplier is used to determine whether the multiplicand is added or not. The figure below will demonstrate both decimal and binary multiplications.

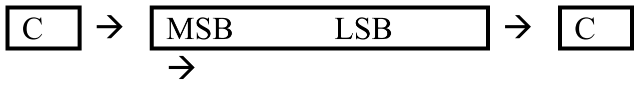
```
Decimal:             23     ← Multiplicand
              *      16     ← Multiplier
              ----------
                     138    ← 6 * 32 = 138
              +      230    ← 1 * 23 = 23 and then shift 23 left one = 230
              ----------
                     368    ← Final Multiplication Product
```

Now, we apply this same knowledge to a binary system 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. Below is an example.

```
Binary(4-bit):        1011   ← Multiplicand
                      *      1101   ← Multiplier
     Multiplier       ----------------
     (LSB)  1         0000   1011   ← 1 * 1011 =1011; 1011 shift left zero = 00001011
            0         0000   0000   ← 0 * 1011 =0000; 0000 shift left one  = 00000000
                      ----------------
                      0000   1011   ← Add results together
            1         0010   1100   ← 1 * 1011 =1011; 1011 shift left two  = 00101100
                      ----------------
                      0011   0111   ← Add results together
     (MSB)  1         0101   1000   ← 1 * 1011 =1011; 1011 shift left three = 01011000
                      ----------------
                      1000   1111   ← Add results together to get final product

     Proof:  Multiplicand   Multiplier      Product
     Bin     1011      *     1101     =     1000 1111
     Dec       11      *       13     =       143          ← Correct
```

Although the method above is sound and easily understandable, there is a more efficient method. For the purpose of this example, assume you are running on a 4-bit system where the registers are 4-bits. It would take 4 registers to do the method above. A better way would be to share the low result registers with the multiplier registers so you can shift all the registers at once and only use 3 registers. In order to do this, you would shift everything to the right instead of to the left and also rotate through the carry. So if the carry bit is set after the shift, add the multiplicand to the high register of the result and shift again; otherwise, just shift. Note that when shifting to the right, you need to rotate through the carry; this means that whatever is in the carry bit will be shifted in the most significant bit and the least significant bit will be shifted out into the carry bit. For example:



Rotate Right Through Carry Bit

The following figure demonstrates this:

```
Multiply                    1011   ← Multiplicand
                 *          1101   ← Multiplier
           Carry  ---------------
                  0000      1101   ← Load Multiplier into low register
           1      0000      0110   ← Shift right through carry
                  1011             ← Carry is set, so add multiplicand
                  ----------------
           0      1011      0110   ← Result of addition
           0      0101      1011   ← Shift right through carry
                  -----            ← Don't add since carry is 0
                  ----------------
           0      0101      1011   ← Result thus far
           1      0010      1101   ← Shift right through carry
                  1011             ← Add multiplicand since carry is set
                  ----------------
           0      1101      1101   ← Result of carry
           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, the shift and add technique can be easily created with a simple for loop that loops a specific amount of times depending on the size of the data, for the case above we used 4-bit numbers so we looped 4 times. In the loop there is a simple rotate left and an add depending on the carry bit. This technique can be easily used for any sized number with minimal effort and is used internally for multiplications with most micro architectures.