Lab 5 – Large Number Arithmetic

SECTION OVERVIEW

Complete the following objectives:

- Understand and use arithmetic and ALU instructions.
- Manipulate and handle large numbers.
- Create and handle functions and subroutines.
- Verify the correctness of 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 Architecture manual. If you consult any online sources to help answer the questions below, you must list these sources 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 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. 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.

Multi-byte Addition

The following example demonstrates the addition of two 16-bit numbers. One of the 16-bit numbers is located in registers R1:R0 and the other number is in R3:R2. The result of this 16-bit operation is placed in registers R6:R5:R4.

Possible Carry-out of R0 + R2 Addition -> 1

R1 R0
Possible Carry-out of R1 + R3 Addition -> 1 R3 R2
+ ------------
R6 R5 R4

As this calculation demonstrates, we first need to add the 8-bit values in R0 and R2 together. The result of this addition is stored in R4. Next, we add the contents of R1 and R3 together, and accommodate for any carry-out bit from the previous operation. The result of this second addition is stored in R5. Finally, if the result of this second addition generates a carry-out bit, then that bit must be stored in R6.

Why is the third result register (R6) necessary? Since we are preserving the carry-out bit from our 16-bit addition, we actually get a 17-bit result: 16-bits with a new MSB in front, which is the possible carry-out bit. However, since the ATmega128’s 8-bit AVR architecture inherently uses general purpose registers that are only 8 bits wide, we must use a third 8-bit register just to store the 17th bit of the addition result. So, we have essentially generated a 24-bit result,
where the most significant byte only has a value of either 1 or 0 (depending on if there was a carry-out bit).

Multi-byte Subtraction

Subtraction of one 16-bit number by another 16-bit number works similarly to the 16-bit addition. The first step is to subtract the low byte of the second operand from the low byte of the first operand. Then, subtract the high byte of the second operand from the high byte of the first operand, making sure to properly account for a 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 also we will assume that the subtraction result will be positive (i.e., the first operand’s magnitude will be greater than the second operand’s magnitude).

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 most likely learned to use for multiplying 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 parts, perform a series of 8-bit 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
-------
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 is the low result byte of the A2 * B1 8-bit multiplication.

Finally, observe that the four result bytes are determined 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

For this lab, you must write an AVR assembly program that calculates the expression \((D - E) + F\)^2. D, E, and F are unsigned 16-bit (2 byte) numbers. A skeleton file (ece375-L5_skeleton.asm) has been provided for your convenience. This is a simulation-based lab, which does not require you to actually run your code on your mega128 board.
To keep things relatively simple, you can assume that $D > E$, so that the subtraction’s result is still positive. This has the added benefit of guaranteeing that the subtraction’s result can fit within 16 bits. However, the addition expression $((D - E) + F)$ will still need to have 3 bytes allocated for its result.

When the 24-bit addition result is then multiplied by itself for the last step of the $((D - E) + F)^2$ expression, this will produce a final 48-bit (6 byte) multiplication result.

You will want to write three functions for this lab: (1) a function that performs 16-bit addition and produces a 24-bit result, (2) a function that performs 16-bit subtraction and produces a 16-bit result, and (3) a function that multiplies two 24-bit numbers and produces a 48-bit result. Once these functions are written, then in the main section of your program you can use them to calculate the result of the $((D - E) + F)^2$ expression.

Each of these functions should read its input operands from Data Memory, and also store its result in Data Memory. 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 also easy for your TA to check that your final answer is correct.

The test values you must use for $D$, $E$, and $F$ are provided in the skeleton file (as Program Memory constants). You can load these values from Program Memory, or you can simply enter the values directly into the Data Memory before each function call.

To demonstrate that your program works correctly, you should put breakpoints before and after each function call. Then, your TA can observe the results of your addition, subtraction, and multiplication functions and confirm that they are correct.

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

No additional questions for this lab assignment.

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, it will count for both the regular lab assignment and the challenge portion.

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

\[
\begin{align*}
23 & \leftarrow \text{Multiplicand} \\
\ast & \leftarrow 16 \leftarrow \text{Multiplier} \\
\hline \\
138 & \leftarrow (6 \times 23 = 23 + 23 + 23 + 23 + 23 + 23 = 138) \\
+ & \leftarrow 230 \leftarrow (1 \times 23 = 23, \text{but then shift left once to get 230}) \\
\hline \\
368 \leftarrow \text{Product}
\end{align*}
\]

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:

\[23 \times 16 = (23 \times 6) + (23 \times 10)\]
Example: Binary \textit{shift-and-add}

Binary (4-bit): \[ 1011 \leftarrow \text{Multiplicand} \]
\* \[ 1101 \leftarrow \text{Multiplier} \]

Multiplier ---------

(LSB) \[ 1 \quad 0000 \ 1011 \leftarrow 1 \times 1011 = 1011, \text{then shift left 0 times} \]
\[ 0 \quad + \quad 0000 \ 0000 \leftarrow 0 \times 1011 = 0000, \text{then shift left 1 time} \]

---------

0000 1011 \leftarrow \text{Add results together}
\[ 1 \quad + \quad 0010 \ 1100 \leftarrow 1 \times 1011 = 1011, \text{then shift left 2 times} \]

---------

0010 1111 \leftarrow \text{Add results together for final product}

As you can see, in this example, we've performed the same \textit{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:

\[ 1011 \times 1101 = (1011 \times 0001) + (1011 \times 0000) + (1011 \times 0100) + (1011 \times 1000) \]

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

\[ 1011 \times 1101 = (1011 \ll 0) + (0000 \ll 1) + (1011 \ll 2) + (1011 \ll 3) \]

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, 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 \textit{shift-and-add} (more efficient)

\[ 1011 \leftarrow \text{Multiplicand} \]
\* \[ 1101 \leftarrow \text{Multiplier} \]

Carry ---------

[0000 1101 \leftarrow \text{Load Multiplier into low register}]
[1011 \leftarrow \text{Carry is set, so add multiplicand}]}

---------

0101 0111 \leftarrow \text{Rotate right through carry]

---- \leftarrow \text{Don’t add since carry is 0]

---------

0101 1011 \leftarrow \text{Result thus far}]

1011 \leftarrow \text{Add multiplicand since carry is set]}

---------

1101 1110 \leftarrow \text{Result of addition}]

1011 \leftarrow \text{Add multiplicand since carry is set}]

---------

1000 1111 \leftarrow \text{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 \textit{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 \textit{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.