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.

As we have been through in lab2, due to a compatibility issue with the simulator, **we will again 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.

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

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

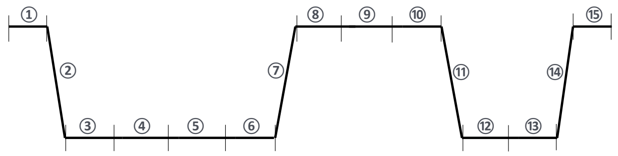

Figure 1: Sample Input to External Interrupt

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`.

LAB REPORT

For every lab, you will be required to submit a write-up report that details what you did and why. As a requirement, your lab report should consist of an Introduction, Design Document, Program Overview, Testing, Questions, Difficulties, and Conclusion. **You must fill the program overview section with a summary of what you have understood by reading/programming comments/code of your AVR code. A detailed description of each section is also required to be elaborated (please refer to the description povided in the report submission page in Canvas). You also need to answer the study questions given in this document.** You must provide your answers using the **"Lab Report Template"** provided in the Canvas webpage. Additionally, you must include your source code at the bottom page of your report. All the submissions including a write-up report and a source code should be done via **Canvas** before the start of the following lab. **NO LATE WORK IS ACCEPTED**.

Note, code that is not well-documented will result in a **severe loss of points** of your lab grade. For an example of the style and detail expected of your comments, look at the provided skeleton code you downloaded. Generally, you should have a comment for every line of code. In terms of Study Questions, some are related to the current lab tasks and some are related to the next lab tasks. As is often the case in engineering courses, there will be some occasions when you are exposed to information that has not been covered in class. In these situations you will need to get into the habit of being proactive and using your study skills to research the answers. This can involve strategies such as reading ahead in the textbook, checking the datasheet, searching online, or reviewing other documentation from the manufacturer.

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?
3. In computing, there are traditionally two ways for a microprocessor to listen to other devices and communicate: *polling* and *interrupts*. Give a concise overview/description of each method, and give a few examples of situations where you would want to choose one method over the other.
4. Describe the function of *each bit* in the following ATmega32U4 I/O registers: `EICRA`, `EICRB`, and `EIMSK`. **Do not** just give a brief summary of these registers; give specific details for each bit of each register, such as its possible values and what function or setting results from each of those values. Also, **do not** just directly paste your answer from the datasheet, but instead try to describe these details in your own words.
5. The ATmega32U4 microcontroller uses *interrupt vectors* to execute particular instructions when an interrupt occurs. What is an interrupt vector? List the interrupt vector (address) for each of the following ATmega32U4 interrupts: Timer/Counter0 Overflow, External Interrupt 6, and Analog Comparator.
6. Microcontrollers often provide several different ways of configuring interrupt triggering, such as *level detection* and *edge detection*. Suppose the signal shown in Figure 1 was connected to a microcontroller pin that was configured as an input and had the ability to trigger an interrupt based on certain signal conditions. List the cycles (or range of cycles) for which an external interrupt would be triggered if that pin's sense control was configured for: (a) rising edge detection, (b) falling edge detection, (c) low level detection, and (d) high level detection. Note: There should be no overlap in your answers, i.e., only one type of interrupt condition can be detected during a given cycle.