CSE 102 – Studio 5

Assembly Language

We will use the inline assembly language style introduced in class to write all of the code for this studio. Recall the form for inline assembly is as illustrated below:

Declarations are in C

```
byte a;
int x;
```

If I wished to write assembly code that is equivalent to the following C code

```
a = 10;
x = a;
```

I would author the following inline assembly code

```
asm volatile(
  "ldi r24,10    ;load immediate the value 10 into r24 \n\t"
  "sts (a),r24   ;store r24 into memory location a \n\t"
  ::: "r24"
);
asm volatile(
  "lds r24,(a)   ;load memory location a into r24 \n\t"
  "ldi r25,0     ;load immediate the value 0 into r25 \n\t"
  "sts (x),r24   ;store lsByte of (r25:r24) into x \n\t"
  "sts (x+1),r25 ;store msByte of (r25:r24) into x+1 \n\t"
  ::: "r24","r25"
);
```

Each `asm` command above corresponds to an individual line in the original C code. This is not required, but breaking up the assembly into multiple `asm` commands facilitates debugging.

Also, when using inline assembly language, you are limited to using registers `r16` to `r31`. You can read all the registers, but you can only write to those above `r15`.

In general, global integer variables defined in C as `int var;` may be loaded using

```
lds r16,var
lds r17,var+1
```

The `gcc` assembler allows the memory address for the `lds` instruction to be in parens `(var)` or not. The same goes for the store instruction, `sts`.

You may need to add the `volatile` keyword in front of the declaration to ensure that the compiler doesn't optimize the variable out of existence. E.g., `volatile int var;`

The code at this link defines a routine `printRegs()` which will safely print the contents of the processor's registers to the PC's screen. You can load it into the Aruduino environment in the same way you added `font.h` in Assignment #4, only in this case you do not need to provide an `#include` statement, as the linker will find the routine as long as you declare it in your program (prior to use):

```
void printRegs(void);
```

The full manual for the AVR assembly language instructions is available here. It includes descriptions of the addressing modes as well as the instructions themselves.

Data manipulation

Starting with the following declarations

```
int m;
int n;
int p;
```

Write assembly language code to do the following equivalent C statements

```
m = 100;
n = m + m + m;
p = m – 50;
```

Make sure to explore the use of the add- and substract-with-carry instructions (`adc` and `sbc`) to handle the 16-bit arithmetic correctly. What registers did you use to store `x`, `y`, and `z`? Use `printRegs()` after each "equivalent C statement" to ensure that the register file contains what you think it does. Use `Serial.print()` or `Serial.println()` to ensure that `x`, `y`, and `z` contain what you think they should.

Control Flow

Starting with these declarations

```
byte a = 0;
byte b = 1;
byte c = 0;
byte d = 0;
```

Write assembly language code to perform the following equivalent C statements

```
if (a == b) {
    c = 1;
}
```

A good idea is to follow the template below:

```
        [insert code to evaluate conditional and put result in SREG]
        brne endif1    ;note test is opposite of if statement
        [insert code for true clause]
    endif1:
        [insert following code]
```

Also, any time you include a label (e.g., the label `endif1` in the above template), start a new `asm` command in your inline assembly code

```
asm volatile(
  "endif1: \n\t"
  :::
);
```

Once you are convinced that your code above works (both with the initial declarations and with a test case in which a and b are equal), add the following in assembly language

```
if (a > b) {
    d = 2;
}
else {
    d = 3;
}
```

Test this code with several values for a and b. Again, use `printRegs()` to make sure things are happening as you think they should.

Next, change your control flow code above to accommodate a, b, and c being defined as integers. This requires not only that you properly handle 16-bit values, but integers are signed, while bytes are unsigned.

Arrays and Indexing

The next challenge is use indirect addressing to access an array. For this exercise, we'll keep it simple and stay with byte-sized individual data elements.

Declare the following in C

```
byte sum = 0;
byte val[8] = {0,5,2,7,16,24,2,1};
```

Write assembly language to perform the following

```
for (i=0; i<8; i++) {
    sum += val[i];
}
```

You do not need to allocate an explicit variable `i` in your assembly code, you may keep it in a register.

There are two challenges to this exercise. You need to control the loop, ensuring that it is executed the correct number of times, and you need to access the array `val`, which you should do using one of the indirect addressing modes.

To access the array, load the base address (the address of `val`) into one of the register pairs that supports indexed addressing modes, X (`r27:r26`), Y (`r29:r28`), or Z (`r31:r30`) and add the `index`. Since addresses are 16 bits, we need to make sure that the index addition supports any carry that goes from bit 7 (in the low byte) to bit 8 (in the high byte).

```
ldi r30,lo8(var) ;use ldi for a pointer, lo8 and hi8 are macros
ldi r31,hi8(var)
ldi r16,index    ;if index isn’t already in a reg, put it there
add r30,r16
adc r31,r1       ;r1 is always zero in compiled C code
ld  r17,z        ;actually does the load of the array val
```

Also look at the post-increment addressing mode, Figure 8 in the manual, as it can simplify things for moving to the next array index value.

Make sure you test your code by altering the initial contents of the array `val`.

Before you go, show off your assembly language to an instructor or TA.