Name: Lab Section:

CprE 288 Fall 2012 – Homework 11
Due Thu. Nov. 29 in the class

Notes:

- **Start early on homework.**
- Homework answers must be typed using a word editor. Hand in a hard copy in the class, before or at the end of the lecture.
- Late homework is accepted within three days from the due date. **E-mail late homework to both of the grading TAs, Min Sang Yoon (my222@iastate.edu) and Zhen Chen (zchen@iastate.edu)**. *Late penalty is 10% per day (counting from 10:45am of the due date if you are in the morning class or 2:10pm if you are in the afternoon class).*

Question 1: Read the following sections of the indicated documents located on the CPRE 288 course page and answer the following questions. Note: these documents will be useful throughout the homework [15 pts]

- Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors
  - Section 1 (1 page)
  - Section 2 (4 pages)
  - Section 5 (4 pages)
  - Section 6.7 (1 page)
  - Section 7 (3 pages)
- http://class.ee.iastate.edu/cpre288/resources/docs/doc2467.pdf (Atmega128 Datasheet)
  - Pages 9 – 13
- ATMega128 Starter Guide
  - Section 4.1 – 4.1.2.2
- AVR Assembler Guide
  - Section 4.4 "Instruction Mnemonics" (5 pages)
- http://class.ee.iastate.edu/cpre288/resources/docs/doc1234.pdf (Mixing C and Assembly Code)
  - Page 6
- Review lecture slides

Question 1: Review the following sections of the indicated documents located on the CPRE 288 course page and answer the following questions. Note: these documents will be useful throughout the homework [20 pts]

- Become familiar with what type of information is contained in "AVR Instruction Set Manual"
- Beginners Introduction to the Assembly Language of ATMEL-AVR-Microprocessors
  - Section 1 (1 page)
  - Section 2 (4 pages)

Name: Lab Section:

- Section 5 (4 pages)
- Section 6.7 (1 page)
- Section 7 (3 pages)

- http://class.ee.iastate.edu/cpre288/resources/docs/doc2467.pdf (Atmega128 Datasheet)
  - Pages 9 – 13
- ATMega128 Starter Guide
  - Section 4.1 – 4.1.2.2
- AVR Assembler Guide
  - Section 4.4 "Instruction Mnemonics" (5 pages)
- http://class.ee.iastate.edu/cpre288/resources/docs/doc1234.pdf (Mixing C and Assembly Code)
  - Page 6
- Review lecture slides

a) Give a short example of an assembly function that uses a globally defined C variable. Show the keywords needed in both the C code and assembly code for your example. [4 pts] (Hint see page 6 of "Mixing C and Assembly Code)

```
#include <avr/io.h>

char max;
char a, b;
extern void asm_max(void);

void main(void)
{
    asm_max();

    while (1)
        {}
}

; assembly code
#include <avr/io.h>

.global asm_max

; external symbols that are used
.extern max

.text

; copy the max of a and b into max
; reg usage: a in r24, b in r22
asm_max:
          LDS        r24,  a          ; load a
```

Name: Lab Section:

```
            LDS         r22,  b           ; load b
            CP          r24,  r22         ; cmp a, b
            BRLT        else              ; br if a<b
            STS         max,  r24         ; max = a
            RJMP  endif
else: STS   max, r22          ; max = b
endif:
```

Grading: It doesn't have to be this long.

b) Why is the "q" parameter in `LDD Rd, Y+q` constrained to be between 0-63? [2 pts]

Only 6 bits is allocated to the "q" parameter in the binary encoding of the LDD instruction, so its range is limited to 0-63.

c) What is the width (in bits) of the ATMEGA128's code memory, and why? [2 pts]

The code memory is 16-bit wide, and ATMega128 is a RISC-type processor. Therefore, most instructions are 16-bit, and some are 32-bit.

d) What is the width (in bits) of the ATMEGA128's Data memory, and why? [2 pts]

The data memory is 8-bit, as the CPU is 8-bit and all the registers are 8-bit. (There is no advantage of using 16-bit memory.)

e) Most ATMEGA instructions can be encoded in 2 bytes (i.e. 16-bit). Why does the encoding of `LDS Rd, k` require 4 bytes? [2 pts]

The memory address of k requires 16 bits. So the instruction LDS Rd, k would not fit in a 2-byte (one-word) instruction.

f) Give the general binary encoding for `ADD Rd, Rr` and the specific binary encoding for `ADD R2, R18` [2 pts]

Name: Lab Section:

General binary coding: `0000 11rd dddd rrrr`
Specific binary encoding: `0000 1110 0010 0010`

g) Give the general binary encoding for `LDS Rd, k` and the specific binary encoding for `LDS R1, 0x40FF` [2 pts]

General Binary coding: `LDS Rd, k`
```
1001 000d dddd 0000
Kkkk kkkk kkkk kkkk
```

Specific Binary Encoding: `LDS R1, 0x40FF`
```
1001 0000 0001 0000
0100 0000 1111 1111
```

h) For what purpose are the registers `X, Y,` and `Z` used that register R1 – R25 cannot be used? [2 pts]

X,Y,Z are used as 16-bit pointer registers to access memory locations in SRAM. They are used to load and store directly. R1-R25 cannot be used, as LD/ST instructions do not work with these registers.

Question 2 (Assembly Practice) [30 pts].

Notes: For programming exercises, points may be deducted for following reasons:

- The program does not compile
- The program does not produce the correct output.
- The program does not follow good programming style, including commenting, indentation and variable naming.
- The program is obviously more complex than necessary

Questions a-f: Assume that you are writing the assembly code for an assembly function called "asm_func". It is called from the main function in the following C program file called "test-main.c":

```
#include <avr/io.h>
#include <stdio.h>
```

Name: Lab Section:

```
signed char ch1 = 2;
signed char ch2 = -3;
signed char flag;
int a = 0x10FF;
int b = 0x80F0;

signed char *pch = &ch2;
int *pint = &b;

void asm_func();

int main()
{
     asm_func();
}
```

Function `asm_func()` should be in a separate assembly program file called "test-asm.S", whose template is as follows:

```
#include <avr/io.h>

.global asm_func

.extern ch1 ch2 flag a b pch pint

asm_func:
     ;
     ; YOUR CODE SHOULD BE PUT HERE
     ;
     ret
```

Name: Lab Section:

Create a project and add the two files in AVR Studio 5. Test your assembly code on the AVR simulator with ATmega128 as the device. When you create the project, choose "AVR GCC" as the project type (if you choose AVR Assembly, the AVR assembler would be called, which is somewhat different from the GCC assembler).

Use complier optimization "-O0" (under menu "Project" => "Configuration Options") to help debug.

After you build the project, start "Debugging", open the watch window (from menu "view"), and add the global C variables into the watch list. Watch how the values of those variables change to verify your assembly functions work.

The `asm_func()` should perform the following operation. Each question is independent.

a. [5 pts] Copy variables using a pointer

```
   *pch = ch1;

; We need both pch and ch1 in registers before saving the value ch1
; to *pch
LDS   r30,  pch        ; load pch to Z-reg
LDS   r31,  pch+1
LDS   r24,  ch1        ; load ch1
ST    Z,    r24        ; *pch = ch1
```

b. [5 pts] Read data from memory using a pointer

```
   a = *pint;

   ; First load pint, then load *pint
   LDS      r30,  pint       ; load pint to Z-reg
   LDS      r31,  pint+1
   LD       r16,  Z+         ; load *pint
   LD       r17,  Z
```

Name: Lab Section:

```
STS     a,    r16        ; a = *pint
STS     a+1,R17
```

c. [5 pts] initialize a pointer

```
pint = &b;

; It's the same as saving a 16-bit constant to a
; memory variable
LDI     r24, lo8(b)
LDI     r25, hi8(b)
STS     pint, r24
STS     pint+1, r25
```

d. [5 pts] Data operations: multiplication

```
a = ch1 * ch2;

; Use MULS for signed type 8-bit multiplication
; The result of MULS is stored in r1:r0
LDS  r16,     ch1        ; load ch1
LDS  r17,     ch2        ; load ch2
MULS r16,     r17        ; ch1*ch2
STS  b,       r0         ; store to b
STS  b+1,     r1
```

e. [5 pts] Data operation: bitwise

```
ch1 = ch1 & ch2;

; & is bitwise AND, use the AND instruction
LDS     r24,  ch1        ; load ch1
LDS     r25,  ch2        ; load ch2
AND     r24,  r25        ; ch1&ch2
STS     ch1,  r24        ; store ch1
```

Name: Lab Section:

f. [5 pts] Testing if two expressions are equal

```
   if ((a - b) == 8)  // 8 is 0x0008 in 16-bit
   {
     flag = 1;
   }
```

```
      ; r25:r24 for a, r23:r22 for b
      LDS   r24,  a          ; load a
      LDS   r25,  a+1
      LDS   r22,  b          ; load b
      LDS   r24, b+1
      SUB   r24,  r22        ; a - b
      SBC   r25,  r23
      CLR   r1               ; r1 = 0
      CPI   r24,  8          ; compare a-b with 0x0008
      CPC   r25,  r1
      BRNE  endif            ; branch is the condition is false
      LDI   r24,  1          ; flag = 1
      STS   flag, r24
endif:
```

Question 3: Predict the execution time. The following is an assembly implementation of function strcpy(). Assume that the length of the input string "src" is 30 characters plus the NULL characters. How many CPU cycles does the execution of the function take on ATmega128? Give a precise number, and explain. [10 pts]

You may find the latency of instructions in ATmega128 datasheet, from pages 365-367.

Hint:

1. **A branch instruction has a short latency if the branch is not taken, and a long latency if the branch is taken.**
2. **You may confirm your answer by testing the function in the AVR Studio Environment.**

```
; C prototype: void strcpy (char *dst, char *src)
; dst in R25:R24, src in R23:R22
.global strcpy
strcpy:    movw r30, r24    ; Z<=dst
```

Name: Lab Section:

```
            movw r26, r22    ; X<=src
loop:       ld   r20, X+      ; ch=*src++
            st   Z+, r20      ; *dst++=ch
            tst  r20          ; ch==0?
            brne loop         ; loop if not
            ret
```

The loop body will be executed 31 times. The “brne” branch is taken for first 30 iterations and not-taken for the last iteration (when the NULL character is seen).

Total number of cycles = 1 + 1 + (2 + 2 + 1 + 2) * 30 + (2 + 2 +1 + 1) + 4 = 222 cycles.

Grading: Give appropriate partial credit.