ECED3204 – Lab #2

STUDENT NAME(s): _____ .

STUDENT NUMBER(s): B00 _____ .

Pre-Lab Information

It is recommended that you read this entire lab ahead of time. Doing so will save you considerable time during the lab, as you will be required to write some simple C code during this lab!

Overall Objective

This lab has several main objectives:

- Learn about use of in-line assembly
- Learn about generation of listing files
- Compare optimization levels

Part #1: Use of In-Line Assembly

Objective

- Familiarize yourself with use of in-line assembly

Required Materials

- Computer with Atmel Studio 6.2

Background

In Lab #1, you used both Assembly and C to program a microcontroller. This lab will introduce you to the use of mixing assembly and C into a single file.

Procedure

1. Start a new C/C++ project (see Lab #1 for details), with the following contents:

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

int main(void)
{
   DDRB |= 1<<0;
   while(1){
          PORTB |= 1<<0;
          //TODO: need a delay here?
          PORTB &= ~(1<<0);
          //TODO: need a delay here?
   }
}
```

2. Build the project, and confirm it builds OK. We now want to insert a delay such that the toggling of PORTB.0 happens at a slightly slower rate. To do so we'll use in-line assembly, which allows us to insert assembly code into the C code.

   To do this, we use the following code fragment:

```
asm volatile("nop"::);
```

   This will insert the assembly directive "no operation". Insert this into the two spots marked as requiring a delay in the C code.

3. Build the example application, and confirm the code is built OK.
4. How do we confirm the code is performing the operation we require? Perform either a simulation of the code from within Atmel Studio, or download the code to a physical device. What frequency does the PORTB.0 pin toggle at?

Part #2: Reading Listing Files, Changing Optimization Settings

Objective

- Familiarize yourself with the compiler outputs.

Background

The compiler generates a variety of useful files. These can help you understand what is happening to your code, and this part of the lab will teach you about these files.

The main file of interest is the Listing file, which has the extension .LST or .LSS. It is important to always confirm what the compiler is doing when you see 'odd' behaviour.

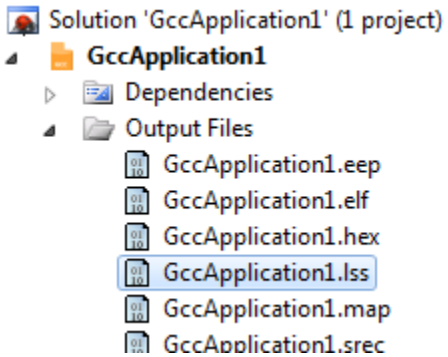
Procedure

1. Start a new C/C++ project, and add the following to the main file:

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

int read_button(void)
{
    return PINB & (1<<0);
}

int do_math(int a, int b)
{
    int c;
    c = a + 7;
    c *= b;
    return c;
}

int main(void)
{
    DDRD |= 1<<0;
   while(1){
          if (read_button()){
             PORTD |= (1<<0);
         } else {
             PORTD &= ~(1<<0);
         }
         PORTC = do_math(4, 10);
   }
}
```

2. Confirm the above code builds. Record the "Program Memory Usage" (Flash) and "Data Memory Usage" (SRAM) usage.

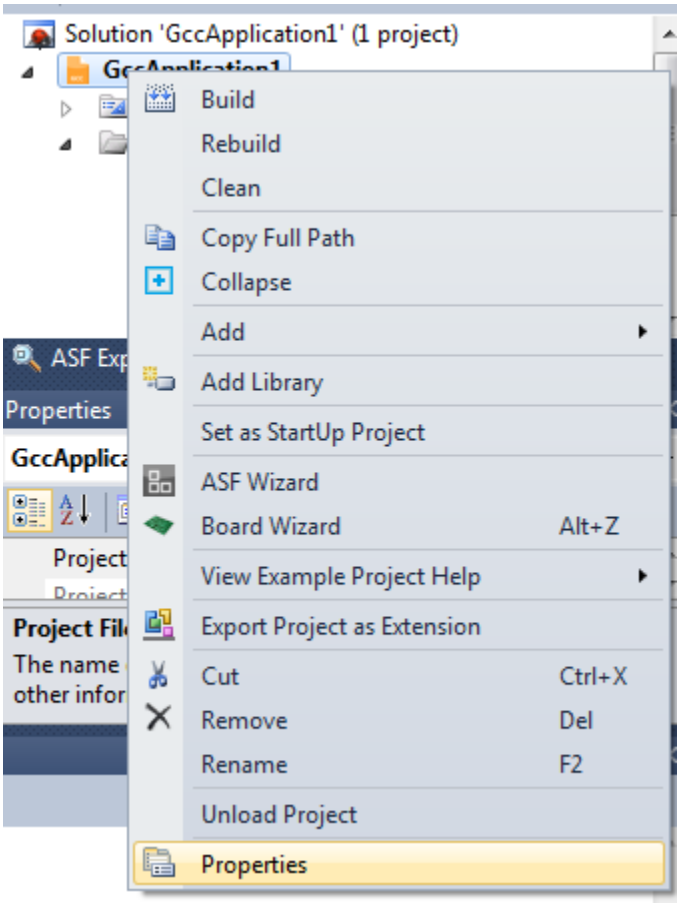
3. Open the Listing file, which is found in the "Output Files" section of the "Solutions Explorer". Remember the Listing file has the ".lss" extension:



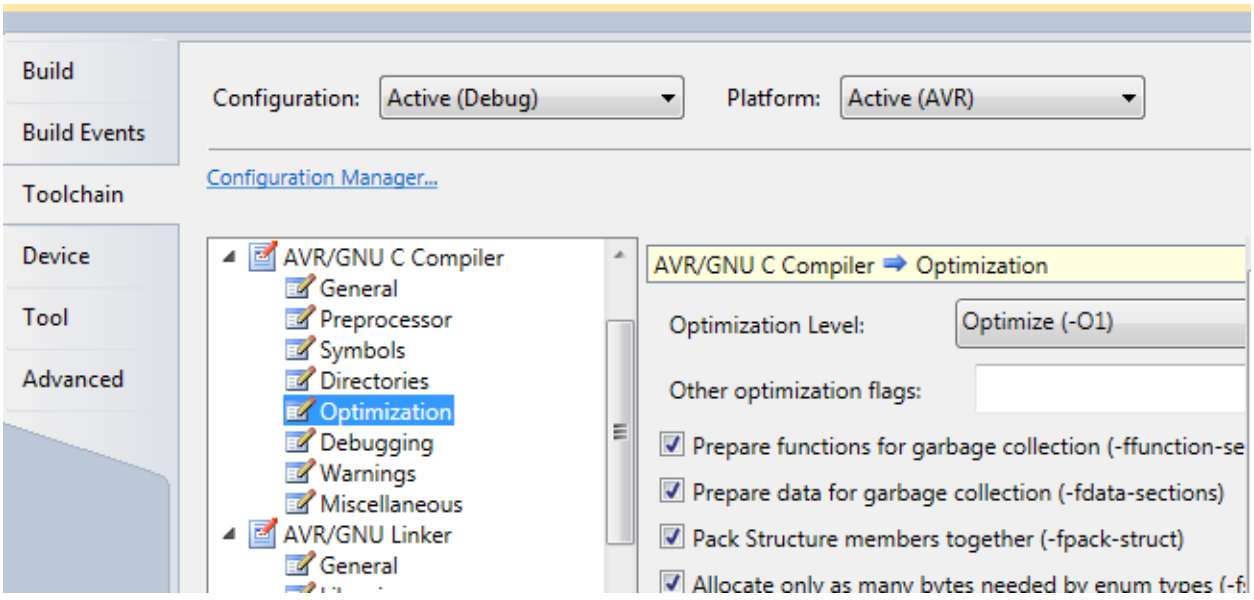
4. Scroll down the file, and find your C source is now interspersed with the resulting assembly code:

```
int main(void)
{
    DDRD |= 1<<0;
    while(1){
        if (read_button()){
  98:   18 9b           sbis    0x03, 0 ; 3
  9a:   02 c0           rjmp    .+4         ; 0xa0 <main+0xc>
            PORTD |= (1<<0);
  9c:   58 9a           sbi 0x0b, 0 ; 11
  9e:   01 c0           rjmp    .+2         ; 0xa2 <main+0xe>
            } else {
            PORTD &= ~(1<<0);
  a0:   58 98           cbi 0x0b, 0 ; 11
        }
        PORTC = do_math(4, 10);
  a2:   88 b9           out 0x08, r24   ; 8
    }
  a4:   f9 cf           rjmp    .-14        ; 0x98 <main+0x4>
```

5. Note in particular the following details:
   a. The set and clearing of bits has been replaced with the “sbi” and “cbi” instructions.
   b. The call to do_math() appears missing – why is this? Instead it is set to the value held in register “r24”. Look for a “ldi” instruction which sets the value of “r24” earlier in the code.
6. Right-Click on the application, and select “Properties”:



7. Switch to the "Toolchain" → "Optimization" option:



8. Change the "Optimization Level" to "None (-O0)", save the project, and re-build it.
9. Record the "Program Memory Usage" for the new build.

10. Compare the resulting assembly listing file between the "-O0" and "-O1" levels. See Lab Questions for specific things to look for.

Lab Questions

1. In Part #1, what was the output frequency? What happens if you add more "nop" instructions?

2. What was the "Program Memory Usage" for "-O1", "-O2", and "-O0" optimization levels?

3. Compare the code around the call to "do_math()" – with optimization turned off, note there is a "rcall" instruction. This instruction is calling the "do_math()" subroutine as expected. But with any level of optimization that disappears. What happened to the subroutine call, and where did the value being loaded instead come from?

   HINT: The optimizer will eliminate calculations that can be done at compile time, since the inputs are constant. Consider what do_math() does, and calculate what you expect the output value to be.

4. With optimization enabled, setting and clearing a PORT bit collapses to a single instruction (sbi or cbi). With optimization set to "None", what instructions are used to perform this task?