CECS-412-01

Summer 2018

Project #1

Name	Report (20 Points)	Demo (15 Points)	Quiz (5 Points)
Andrew Nguyen Vo			

INVESTIGATION OF BASIC AVR ASSEMBLY FUNCTIONALITIES

Andrew Nguyen Vo

CECS-412-01

ABSTRACT

During this investigation, the instruction set, directives, and coding structure of AVR Assembly language is explored, which is specific to the microcontroller ATmega328(B). Additionally, some basic debugging and simulation functionalities of the Atmel Studio 7.0 IDE are utilized in the analyses of written code to ensure desired operations are properly executed. Given samples of test code are entered into the IDE and carefully examined line-by-line using the "Step Into" tool of Atmel Studio 7.0's simulated debugging mode. Referencing the AVR Assembly documentation, the expected results of each instruction are compared to the actual results from executing the code. This is done in part by observing the changes and effects on the memory and registers using the IDE view panes. It is concluded by the investigation that an accurate exploration of the AVR Assembly language was achieved as the actual results were executed as expected according to the documentation.

Body

This investigation was conducted using the version of Atmel Studio 7.0 configured for compatibility with Windows. The first set of given code served to explore a number of the directives available in the AVR Assembly language. These directives are prefixed by a period character and belong to a single column within the code. Some directives are used to specify which type of memory the code should target, such as .CSEG for code segments in FLASH memory, .ESEG for EEPROM segments, and .DSEG for data segments in SRAM. None of the segment directives are given a parameter. The .ORG directive can be used directly after each segment directive to specify the location counter of each segment in the targeted memory based on the parameter it is given. The .EXIT directive instructs the Assembler to cease assembly of the file. The .DW and .DB directives are used in EEPROM or FLASH memory to define constant words or constant bytes, respectively, by taking an expression or list of expressions as parameters. The .BYTE directive is used to reserve memory resources for variables in SRAM only.

In building the assembly project, various output files are generated, including .hex, .lss, .map, and .obj. .hex is a file format which stores the assembly language code in hexadecimal form that is dumped into the microcontroller to execute. The lines in a .hex file follow a very specific structure according to the format :CCAAAATTXXXXX.....XXSS. Every line in the file is preceded by a colon followed by the character count (CC) which indicates the total number of bytes of data in that line. This is followed by the memory address (AAAA) where the data in this line needs to be dumped. The next two digits (TT) indicate if this is the last line of the code. TT=00 means that the code is not complete, and TT=01 indicates the end of the code. XXXXX.....XX represents the data bytes that are to be dumped into the memory, and the last two digits (SS) is the checksum byte of that line. The .lss file contains a copy of the written source code in addition to offset addresses of the variables and procedures pertinent to the program and the microcontroller. The .map file contains the information about the link that is useful in monitoring and analyzing the sizes of the functions and data in the program. The .obj file contains the offset addresses of data and the contents of that address along with some debugging information.

After the program was built, each line was examined using the “Start Debugging and Break” and “Step Into” functions of the debugging tools. One such line was the instruction “ldi r30,56”, which caused register 30 to contain the value 0x38. With reference to the AVR documentation, the instruction LDI refers to “load immediate”, which loads some constant data into a specified register. It is given two parameters, r30 and 56, which are the register and constant data, respectively. Thus, the register r30 is loaded with the content 0x38, which is the value 56 in hexadecimal form (8 * 16^0 + 3 * 16^1).

Following the execution of this instruction, the program counter is incremented from 0x00000000 to 0x00000001. This is because the program counter is the register that stores the address of the instruction that is currently being executed. Since the “ldi r30,56” has been completed, the register is moved to

the address of the next instruction in the order to be executed, which is one greater than the previous. The yellow arrow in the IDE corresponds to this instruction. As each instruction is executed, the yellow arrow automatically moves to the next instruction in the written code that will be executed, regardless of where it may be. When a red breakpoint is marked on a line and the program is executed using the "Start Debugging" tool rather than the "Start Debugging and Break" tool, the yellow arrow will immediately travel to the first breakpoint line, indicating that the program has halted immediately before this line, and that the breakpoint line is the next to be executed after continuing the program.

As discussed, Atmel Studio 7.0 offers a variety of native debugging tools that are useful in analyzing the program and resolving issues. When executing the code using the "Start debugging…" tools, breakpoints allow the program to run up until that point where it is halted until manually resumed. This allows the programmer to examine and analyze the data and effects of the program until that point. The IDE also offers code stepping functionalities. "Step Into" and "Step Over" allow the programmer to examine the code line-by-line, with the latter executing all the lines in a function altogether if applicable and halting at the next line after the function. The "Step Into" tool would simply go into the function and examine it line-by-line. In contrast, the "Step Out" tool can only be used within a function that has been entered using the "Step Into" tool. The "Step Out" executes the remaining lines of the function and halts at the next line after the function. The "Call Stack" shows where in the code each function should return upon completing. The "Memory" pane allows observation of the contents at each address in various parts of the memory. The "Watch" pane allows tracking of the changes in a register or variable as each line is executed. The "Processor Status" pane allows tracking of the changes in the "Program Counter", "Stack Pointer", variable registers (X, Y, Z), "Status Register", "Cycle Counter", "Frequency", and "Stop Watch".

The code referenced as "Arithmetic, logic, Data Transfer Instructions" in the Source Code section of this document introduces different addressing modes for the specified instructions. LDI, CLR, and SER use the "Register Direct, Single Register Rd" addressing mode as they each only address a single register in FLASH memory. ADD, SUB, AND, and MUL all use "Register Direct Two Registers Rd and Rr" as they all still only address within FLASH memory, but the values of two registers are used. ST uses the "Data Indirect" addressing mode as it stores the value in an address indirectly referenced in SRAM by the X-, Y-, or Z- index. JMP uses "Direct Program Addressing" as it unconditionally continues at the address immediate in the instruction word.

During the execution of the same referenced code, the value at address 0x100 is set to 18. This is because the address 0x100 is indirectly referenced by the X-register, and the ST instruction stored the value of register 30 at this address. From the executed lines of code before the ST instruction, the value 0x18 was generated into register 30 after a series of operations. This hexadecimal value is abbreviated to 18 in the memory pane, which is what is stored at 0x100.

Additionally, the status register bits were changed during the execution of the program. The half carry flag bit and zero flag bit were changed upon execution of the ADD and CLR instructions, respectively. The half carry flag was set because there

was an overflow from one digit to the next during the ADD operation, and the zero flag was set because the value of a register was set to zero in the CLR operation. Upon initially being set, the flag in the status register is colored red to denote that a change has just taken place. Subsequent steps through the code turn the flag bit into a default gray if the flag has been unaffected. Immediately upon being cleared, the flag bit will turn white again with a red border to indicate that the flag has just been cleared by the previously executed instruction.

The code referenced as “Conditional Branch, MCU instructions” in the Source Code section of this document also introduces some new instructions with different addressing modes. BRMI and BREQ use “Relative Program Addressing” as they continue the program at the address PC + k + 1 – “PC” being the program counter and “k” being the offset. NOP does not use any addressing modes as it performs no operation.

In the same referenced code, the SUB instruction sets the sign flag bit, the negative flag bit, and the carry flag bit. As a result of the negative flag bit being set, the following BRMI instruction branched to “here:”. If it was not set, the program would execute the next line in the code. The NOP instruction performed no operation in the program both due to the branching from BRMI and by its definition. The BREQ instruction branches to “here:” if the zero flag bit is set, which it is not.

The code referenced by “Bit Instructions” contains some new instructions that have different addressing modes. LSL, LSR, ASR, BSET, and BCLR all use “Register Direct, Single Register Rd” as they all only address a single register in FLASH memory. ST with the “X+” parameter uses the “Data Indirect with Post-increment” addressing mode as it addresses SRAM memory and post-increments the X-register index.

In the same code, the LSL and LSR instructions are logical bitwise operations that shift the bits’ positions one to the left or the right, respectively. ASR also shifts the bits’ positions one to the right arithmetically. The logical and arithmetic operations are identical except for that arithmetic shifting preserves the sign bit. The BSET and BCLR instructions each set and clear the specified bit in the status register, respectively. In this case, the bit was 2, which corresponds to the negative bit.

In the same program, the BRMI instruction did not branch because the negative flag bit was set to 0 by the BCLR instruction one line previously. The value 0x16 (abbreviated 16) is stored at the SRAM locations 0x100, 0x101, and 0x102. The value was initially generated from the series of operations on register 30. Following those, the “st X+,r30” instructions stored this value in the location specified by the X-register (initially 0x100), and then incremented the location value by 1 after each ST line. This operation was repeated three times, which resulted in 0x16 being stored in the three SRAM locations.

Source Code (Software)

```
/* * project1.asm *
* Created: 1/18/2018 5:18:16 PM
* Author: Eugene Rockey
*/
//Arithmetic, logic, Data Transfer Instructions
        .cseg                   ;FLASH code segment
        .org    0x0
start:  ldi     r26,0x00        ;Loads the constant hexadecimal 0x00 into register 26 –
                                ;no flags can be set by this operation
        ldi     r27,0x01        ;Loads the constant hexadecimal 0x01 into register 27 –
                                ;no flags can be set by this operation
        ldi     r30,56          ;Loads the constant decimal 56 (hexadecimal 0x38)
                                ;into register 30 – no flags can be set by this operation
        ldi     r31,24          ; Loads the constant decimal 24 (hexadecimal 0x18)
                                ;into register 31 – no flags can be set by this operation
        add     r31,r30         ;adds the value in registers 30 and 31 and places the
                                ;result in register 31 – half carry flag is set but
                                ;potentially all flags can be set by this operation
        sub     r31,r30         ;subtracts the value from register 30 from register 31
                                ;and places the result in register 31 – all flags can be set
                                ;by this operation
        and     r30,r31         ;performs a logical AND on registers 30 and 31 and
                                ;places result in register 30 – the Z, N, V, and S flags can
                                ;be set by this operation
        mul     r30,r31         ;multiplies values from register 30 and 31and places the
                                ;result in register 30 – the Z and C flags can be set by
                                ;this operation
        st      X,r30           ;stores value indirect from register 30 to data space
                                ;using index X – no flags can be set by this operation
        clr     r30             ;sets the value of register 30 to 0x00 and sets the zero
                                ;flag – the Z, N, V, and S flags can be set by this operation
        ser     r31             ;sets the value of register 31 to 0xFF – no flags can be
                                ;set by this operation
here:   jmp     here            ;unconditionally jumps to the address with the
                                ;destination labeled “here:” – no flags can be set by this
                                ;operation
        .exit

/* * project1.asm *
* Created: 1/18/2018 5:18:16 PM
* Author: Eugene Rockey
*/
//Conditional Branch, MCU instructions
        .cseg                   ;FLASH code segment
```

```
        .org    0x0
start:  ldi     r26,0x00
        ldi     r27,0x01
        ldi     r30,56
        ldi     r31,24
        sub     r31,r30
        brmi    here        ;tests the negative flag and branches to the address
                            ;corresponding to the destination “here” if negative is
                            ;set – no flags can be set by this operation
        st      X,r30
        nop                 ;no operation is performed at this line – no flags can be
                            ;set by this operation
        clr     r30         ;this line would have set the value of register 30 to
                            ;0x00, but it is skipped due to the BRMI instruction –
                            ;the Z, N, V, and S flags can be set by this operation
here:   breq    here        ;tests the zero flag and branches to the address
                            ;corresponding to the destination “here:” if zero is set –
                            ;no flags can be set by this operation
        .exit

/* * project1.asm  *
* Created: 1/18/2018 5:18:16 PM
* Author: Eugene Rockey
*/
//Bit Instructions
        .cseg               ;FLASH code segment
        .org    0x0
start:  ldi     r26,0x00
        ldi     r27,0x01
        ldi     r30,0xAC
        lsl     r30         ;performs a logical shift left on register 30 – can affect
                            ;the Z, C, N, V, and H flags
        lsr     r30         ;performs a logical shift right on register 30 – can affect
                            ;the Z, C, N, and V flags
        asr     r30         ;performs an arithmetic shift right on register 30 – can
                            ;affect the Z, C, N, and V flags
        bset    2           ;sets the 2 bit in the status register, which is the
                            ;negative flag – bset can affect any bit in the status
                            ;register
        bclr    2           ;clears the 2 bit in the status register, which is the
                            ;negative flag – bclr can affect any bit in the status
                            ;register
        brmi    here
        st      X+,r30      ;stores the value of r30 in the data space specified by
                            ;the index of the X-register and post-increments the
                            ;index by 1 – no flags can be affected
```

```
        st      X+,r30          ;repeated operation of the previous line
        st      X+,r30          ;repeated operation of the previous line
here:   jmp     here
        .exit

;
; project1.asm
;
; Created: 5/29/2018 11:10:34 PM
; Author : Andrew Nguyen Vo
;

        .cseg                   ;FLASH code segment
        .org    0x0             ;sets the origin to 0x0
start:  ldi     r26,25          ;loads integer 26 into r26: hex(0x19)
        ldi     r27,21          ;loads integer 27 into r27: hex(0x15)
        add     r26,r27         ;adds values of r26 and r27 and stores in r26:
                                ;hex(0x2E)
        sub     r26,r27         ;subtracts values of r27 from r26 and stores in
                                ;r26: hex(0x19)
        mul     r26,r27         ;multiplies values of r26 and r27 and stores high
                                ;byte in r1 and low byte in r0: r1 = hex(0x02)
                                ;r0 = hex(0x0D) decimal = 525
        neg     r26             ;changes value of r26 to be negative: old =
                                ;hex(0x19) new = hex(0xE7)
        ser     r27             ;sets r27 to hex(0xFF)
        clr     r27             ;sets r27 to hex(0x00)
        ldi     r30,0x00        ;sets low byte of z-register to 0
        ldi     r31,0x01        ;sets high byte of z-register to 1
        st      z,r26           ;transfers value of r26 to the indirect SRAM
                                ;location z, which is at address 0x100;
                                ;data(0x0100) = e7
        ld      r28,z           ;transfers data value at z to r28: r28 = hex(0xE7)
        mov     r29,r28         ;copies value at r28 to r29: r28 = hex(0xE7); r29
                                ;= hex(0xE7)
        bset    2               ;sets the negative flag
        brmi    negative        ;branches to the 'negative' part of the code if the
                                ;negative flag is set (it is)
        nop                     ;filler code that does nothing (skipped)
        nop                     ;filler code that does nothing (skipped)
negative:
        bclr    2               ;clears the negative flag
        jmp     continue        ;jumps to the 'continue' part of the code
                                ;unconditionally
        nop                     ;filler code that does nothing (skipped)
        nop                     ;filler code that does nothing (skipped)
continue:
        bclr    0               ;clears the carry bit in the status register
```

```
                bset    0           ;sets the carry bit in the status register
                swap    r29         ;swaps the nibbles of r29: old = hex(0xE7); new
                                    ;= hex(0x7E)
                ldi     r26,5       ;sets value of r26 to integer 5 in preparation for
                                    ;the do-until loop: r26 = hex(0x05)
do:             cpi     r26,1       ;compares the value of r26 to the integer 1
                brlo    until       ;branches to 'until' if the value at r26 is lower
                                    ;than 1
                dec     r26         ;decrements r26 by 1 if code did not branch to
                                    ;'until'
                jmp     do          ;jumps unconditionally to 'do'
until:          nop                 ;does nothing but executes upon completion of
                                    ;the do loop
                ldi     r26,0       ;sets value of r26 to integer 0 in preparation for
                                    ;the for-next loop: r26 = hex(0x00)
                ldi     r27,5       ;sets value of r27 to integer 5 in preparation for
                                    ;the for-next loop: r27 = hex(0x05)
for:            inc     r26         ;increments the value of r26 by 1
                cp      r26,r27     ;compares the value of r26 to the value of r27
                brlo    for         ;branches back to 'for' if the value of r26 is less
                                    ;than the value of r27
next:           nop                 ;does nothing but executes upon completion of
                                    ;the for loop
                .exit               ;exits the code
```

Schematics (Hardware)

None

Analysis

In this investigation, the fundamental instruction set and directives of AVR assembly language is explored through practice and analysis along with the debugging functionalities of the Atmel Studio 7.0 IDE. The operations that were researched and observed are at the core of all embedded systems. It is important to understand how these operations access and affect different parts of the microcontroller memory in order to optimize performance and programming. The branching instructions provide insight into the subroutines of higher level languages and how different parts of the code interact with one another. Using the BSET and BCLR instructions will be essential in setting flags that correspond to different events in the real world that can be detected by an actual embedded system. The skills acquired with the debugging tools will remain applicable in nearly all real-world programming of embedded systems and beyond. Analyzing line-by-line with the step-into tool and observing the code's effects at a controlled rate will facilitate the resolving of bugs and issues in the code.

Conclusion

The purpose of this investigation was to acquire fundamental knowledge of AVR Assembly and basic usage of Atmel Studio 7.0. Based on the findings of this project, it is clear that these essential tools will be necessary for all future programming of embedded systems. Regardless if the programming language is high or low level, it is important to understand that all operations fundamentally communicate with the machine bit by bit through the use of registers and data spaces within the memory. For the ideal optimization of performance, AVR or a similar low-level programming language should be used in managing memory and functions.

References

“AVR Libc Reference ManualA simple project,” *Microchip Technology Inc*. [Online]. Available: https://www.microchip.com/webdoc/AVRLibcReferenceManual/group__demo__project_1demo_project_map.html. [Accessed: 04-Jun-2018].

“Hex File Format,” *3D Printing Processes - Powder Bed Fusion*. [Online]. Available: https://www.engineersgarage.com/tutorials/what-is-hex-file-format. [Accessed: 04-Jun-2018].