CECS/ECE-412
Spring 2022
Lab #1

Drew Branum

Report

(Points 80)

Demo
(15 Points)

Quiz
(5 Points)

Lab 1 Branum

Andrew Branum

ECE 412
Abstract

Assembly programming language is researched and implemented. Specifically, the AVR assembly language using the Atmel 328P(B) MCU. Combining Atmel Studio 7 IDE for windows, the IDE debugger/programmer, the AVR instruction set, and the AVR Assembler User Guide under laboratory conditions, AVR assembly language is implemented to evaluate and experiment with the 328P(B)'s core Programming Model. Test code is inserted into the Atmel Studio 7 IDE, then the code is built and programmed directly into the 328P(B). The Atmel IDE's debugger is utilized to analyze each line of code during its execution within the 328P(B). The data and results are recorded and analyzed. Conclusions are made about the AVR assembly language and 328P(B) based off the data observed in the debugger and the analysis of that data. The conclusion agrees with the Atmel documentation and the examination of Atmel Studio and the 328P(B) with code examples.

Body

The latest Atmel Microchip Studio was installed on the Microsoft Windows PC. After installation, the components of the Studio were explored and tested. A new 8-bit assembler project was created with the ATmega328P(B) and named project1. The main.asm code was automatically produced and was replaced with the following code:

```
        .eseg ;EEPROM data segment, 0x0 to 0x3FF
        .org    0x0 ;program the EEPROM using a .eep file
eevar:  .dw     0xfaff
msg:    .db     "HelloWorld", '\n',0
        .dseg ;SRAM data segment, 0x100 to 0x8FF
        .org    0x100
string: .byte 3 ;a three byte var
        .cseg ;FLASH code segment, 0x0 to 0x3FFF-Loader
        .org 0x0
start:  ldi r30,56
        ldi r31,24
        add r31,r30
here: jmp here
    .exit ;make .exit the very last line
```

The code demonstrates the different types of memory being targeted according to the various assembler directives: .cseg, .eseg, .org, .exit, .dw, .db, .byte, and .dseg. The directives were then researched yielding new information. The CSEG directives “define the start of a Code Segment” (AVR Manual, 4-10), can have multiple in an Assembler file, and have a word, location counter. The ESEG directives are the start of an EEPROM Segment, can have multiple in an Assembler file, and have a byte, location counter. The DSEG directives are the start of a Data Segment, can have multiple in an Assembler file, and have a byte, location. The ORG directives set the location of the relative segment to an absolute value and can be specifically named. The default for the Code and EEPROM counters is 0, and the default for the SRAM is 32 due to the registers using addresses 0-31. The EXIT directive makes the Assembler stop assembling the file. The DW directive allocates memory resources and takes a list of expressions as a parameter delimited by commas with a value in between -32768and 65535. The DW directives must be in a Code or EEPROM segment and should be preceded by a label. The DB directive is the same as the DW directive except that it is 8-bit and the value is between -128 and 225. The BYTE directive allocates memory resources and takes a parameter that determines the number of bytes to reserve in the SRAM.

After researching the various directives, the project was built and new file types, .hex, .lss, .map, and .obj were generated and researched. Hex files contain settings and information that have been saved in hexadecimal file format and can be saved in text or binary formats. LSS files are known as LotusScript Source Code files and contain source code scripts written in LotuScript language. The MAP files map the code to the various registers within the 328P(B). The OBJ file is used for storing 3D models. Once the research was complete, the selected debugger/programmer was changed to ‘Simulator’ instead of using hardware. The debugger was then activated, and the processor, watch, and memory views were evaluated and examined. The ‘Step Into’ icon was then clicked, causing the program counter to update until the code is exited. Stepping through the code, the line ldi r30,56 was executed and the value 0x38 was placed into the r30 register, 56 in hexadecimal: (3x16)+(8x1) = 56. The program counter then incremented by 1 after each time the ‘Step Into’ was clicked. The counter was then set to 0 while still debugging and ‘Step Into’ was clicked again and the counter ended up at 0x00000001 which

is logical considering it reverted to spot 0. Further investigation of the debugger occurred. Breakpoints allow for the code to stop at a specific line, allowing the rest of the program to be evaluated at that point in time. The Debugger also contains 'Step Over' which skips a line of code, and 'Step Out' which steps out of the current code segment.

Arithmetic, logic, and data transfer techniques were coded in assembly language and evaluated as seen in the software section. After stepping through the code, the value at data space 0x100 was examined and yielded 18. The data space contained the value of register 30, as it was set in the program and didn't change after the multiplication command due to registers 0 and 1 holding the product of 30 and 31. The status register bits also underwent examination. When the code was debugged, the components that change during that step are highlighted in red and are black otherwise. Stepping through the code, the status registers bits did the same and were altered once the addition command ran. The register highlighted the half-carry flag which indicated a carry or borrow has been initiated in the addition of two registers. The half carry flag did the same with the subtraction operation. This occurred because the addition and subtraction were in hex, but after they were completed, the flag turned to grey instead of red meaning it isn't being used with the current line of code. The clr command changed the zero flag from white to red meaning that the operation being processed resulted in 0. Those were the only two flags to occur during evaluation.

Branching techniques were then assembled and evaluated within Atmel Studio. The status register bits and SUB instruction were then analyzed. Since register 30 was subtracted from 31, the result was -32 (0xE0) and the sign, negative, and carry bits on the status register were highlighted red. The flags were all set, and the BRMI branched to "here" because the negative flag was set. The NOP instruction stands for no operation and does nothing. The BREQ instruction does not branch because the registers 30 and 31 are not equal causing the program to terminate since it occurred after a SUB operation.

Bit instruction code was next assembled and evaluated in the IDE. The LSL, LSR, ASR, BSET, and BCLR commands were evaluated and tested. Logical shift left shifts all the bits in a given register one place to the left clearing bit 0 and loading bit 7 into the C Flag of the SREG multiplying the value by 2. Logical shift right shifts all the bits in a given register one place to the right clearing bit 7 and loading bit 0 into the C Flag dividing the value by 2. Arithmetic shift right shifts all bits to the right one place but holds but 7 constant and loads bit 0 into the C Flag dividing the value by two without changing the sign. Bit set in SREG sets a single flag within the SREG. Bit clear in SREG clears a single flag in the SREG. BRMI did not branch because the negative flag was not set, it was cleared in the previous line of code. The value 16 was stored in 0x100, 0x101, and 0x102. This value was generated by taking the original value in r30 (172) shifting it left, shifting it right, and arithmetically shifting it right once more. The value ST X+, R30 stored the value in memory location X and incremented the location value by one after executing hence why the value appeared in three separate locations.

Source Code (Software)

```
       .cseg ;FLASH code segment
       .org 0x0
start: ldi r26,0x00 ;loads register 26 with the hex value 0x00, no flags
       ldi r27,0x01 ;loads register 27 with the hex value 0x01, no flags
       ldi r30,56 ;loads register 30 with the hex value of 56 (0x38), no flags
       ldi r31,24 ;loads register 31 with the hex values of 24 (0x18), no flags
       add r31,r30 ;adds registers 30 and 31 and loads the sum (0x50) into
                   register 31, flags Z,C,N,V,H
       sub r31,r30 ;subtracts register 30 from 31 and loads the value (0x18) into
                    register 31, flags Z,C,N,V,H
       and r30,r31 ;preforms logical AND on the registers and puts the result in
                     register 30, flags N,V,Z
       mul r30,r31 ;multiplies unsigned registers 31 and 30 and loads the product
                   into registers 0 and 1 with 1 being the high byte and 0 being
                   the low byte, flags C
       st X,r30 ;stores one byte of data from register 30 to data space location
                 0x100, no flags
       clr r30 ;clears register 30, flags N,V,Z
       ser r31 ;sets all bits (0xFF) in register 31, no flags
stop: jmp stop ;branches to the jmp location (stop) which makes it an infinite
                loop, no flags
       .exit

//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 ;conditionally branches to "here" if the Negative Flag is
                        set, no flags
              st X,r30
              nop ;no operation, no flags
              clr r30 ;clears register 30, flags N,V,Z
here:         breq here ;conditionally branches to "here" if the Zero Flag is set,
                  since it happens after SUB the branch only occurs if the binary
                  number in r30 = r31, no flags
              .exit


//Bit Instructions
              .cseg ;FLASH code segment
              .org 0x0
start:        ldi r26,0x00
              ldi r27,0x01
              ldi r30,0xAC
              lsl r30 ;shifts all bits in r30 to the left one place, bit 0 is
                       cleared and bit 7 is loaded into the carry flag, multiplies
                       the value in r30 by 2, flags C,V,N,Z
              lsr r30 ;shifts all bits to the right one place, bit 7 is cleared and
```

```
                bit 0 is loaded into the C flag, divides value by 2, flags
                C,V,N,Z
        asr r30 ;shifts all bits to the right one place, bit 7 is held con
                stant and bit 0 is loaded into the C Flag, divides value by
                2 without changing the sign, can be rounded, flags Z,C,N,V
        bset 2 ;sets a flag in the status register (2 = Negative), flags
               SREG(x)
        bclr 2 ;clears a flag in the status register, flags SREG(x)
        brmi here
        st X+,r30 ;stores value from r30 into data space location 0x100 and
                  increments location by 1 after completion, no flags
        st X+,r30 ;stores value from r30 into data space location 0x101 and
                  increments location by 1 after completion, no flags
        st X+,r30 ;stores value from r30 into data space location 0x102 and
                  increments location by 1 after completion, no flags
here:   jmp here
        .exit
```

Schematics (Hardware)

NONE

Analysis

Overall, the laboratory process revealed an abundant amount of information. Information about the inner workings of the 328P(B), its registers, memory, and overall operations were learned through research and implementation. The ATMEL datasheet for the 328P(B), User's manual for Atmel, Atmel Studio 7, and google were used to research the topics and devices at hand. After the research was completed, the information was implemented with code examples. These examples provided a general overlook on the basic operations within the 328P(B) and allowed for learning with experience to take place. The different memory locations, registers, and status registers were also included in the evaluation. All the information revealed from the code examples were validated by the research and data sheets.

All of this research and implementation were practiced in order to gain a relatively small amount of knowledge on how embedded systems within micro-computers work. With assembly language being the most commonly used langued to program MCUs, the knowledge gained is useful in various aspect. The information can be applied to numerous real world embedded systems whether it be programming an Arduino to complete tasks automatically or programming a server at work to update its contents automatically at a specific time and date. Assembly language is compiled at the fastest level within machines making it useful. The speed it compiles at allows for precision and accuracy within autonomous machines.

Conclusion

Assembly programming language, Atmel Studio 7, and the 328P(B) were researched and implemented together to create a better understanding on how they work. The Atmel Studio 7 was installed and connected to the 328P(B) where assembly code was executed within the IDE. The execution of the code matched the research from various sources and brought forth supporting data. The Studio was heavily analyzed with all aspects of the debugger being tested and backed up with the data at hand. The code used to test the 328P(B) revealed more data that also matched the data sheets and user manual.

References

Cite all sources you researched and/or used to perform this lab
If no references were used then type 'none'

ATmega328PB Instruction Set Summary. (2021). *ATmega328PB Instruction Set Summary*.

Atmel START User's Guide. (2020). *Atmel START User's Guide*. https://www.microchip.com/content/dam/mchp/documents/atmel-start/Atmel-START-User-Guide-DS50002793B.pdf

LSS File Extension | What is LSS file and how do i open it? (2020). Filex.Com. https://files.tips/extension/lss

FileInfo.com - The File Information Database. (2021). FileInfo. https://fileinfo.com/