ECE-412-01

Fall 2017

Project # 1

|  |
| --- |
| Name |
| Spencer Goff |

Intro to Atmel Studio

Spencer Goff

ECE-412-01

Abstract

The purpose of this project is for me to get acquainted with Atmel Studio and programming in the Assembly language. I’d never used either before this project. To perform this project, I first downloaded Atmel studio. I then used the source code provided (listed below) to run the program. I used the debugger to step through the code and watched the values of registers and SRAM locations change as each command is executed. If I didn’t understand a particular command, I looked it up in the Atmel Studio AVR Instructions datasheet. To perform this project, I used Atmel Studio 7 on a Lenovo Thinkpad Yoga laptop running Windows 8.1 Pro. The purpose of this project was achieved – I now have a basic understanding of how to use Atmel Studio and the Assembly instructions that it provides. I can walk through basic programs and either already know what they’re accomplishing, or figure that out by using the debugger to step through the code and analyze the values in the registers and SRAM locations.

Body

For this project, I used Atmel Studio 7 on a Lenovo Thinkpad Yoga running Windows 8.1 Pro, and did not experience any technical difficulties.

There are many directives used in the source code for this project. Here are there basic meanings:

* .cseg stands for “code segment”, and has its own location counter.
* .eseg stands for “EEPROM Segment”, and signifies the start of an EEPROM segment.
* .dseg stands for “data segment”, which consists of BYTE directives and has its own location counter.
* .org sets the location counter to an absolute value; SRAM location if within a data segment, program location if within a code segment.
* .exit tells the assembler to hault assembling the file.
* .dw reserves resources (in the form of memory) in program memory or EEPROM for constant bytes.
* .db reserves resources (in the form of memory) in program memory or EEPROM for constant bytes.
* .byte reserves memory resources in program memory or EEPROM for a variable.

Many different file types may be used in the scope of embedded software. Some of these include:

* .hex files contain data saved in hexidecimal format. This means numbers are represented in base-16 format.
* .lss files are meant for use by Windows program installers.
* .map files are used by games developed by a Quake engine
* .obj is a 3-D image format that can include coordinates, textures, and more.

One of the instructions in the source code is “ldi r30,56”. This command causes register r30 to contain the value 0x38. This is because 0x38 is a hexadecimal number which is the equivalent of the decimal number 56.

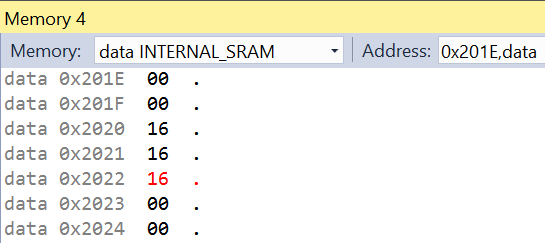
After the program has stopped at the breakpoint and I’ve stepped in 1 line, the program counter been incremented from 0x00000000 to 0x00000001. This is because the program counter increments by 1 for each line that I step into via the debugger. The program counter shows the address of the instruction being executed. It makes sense that the counter incremented from decimal values of 0 to 1 since this was the first instruction that was executed.

While using the debugger to step through the source code, I changed the program counter value to 0 after reaching “here:”, then clicked the “step through” button again. The yellow arrow reset back to the line immediately below the breakpoint. This is opposed to its previous behavior, when it wouldn’t step past the “here:” command once it reached it.

In the beginning of the program, we store 0x20 into registers r26 and r27. During this process, X is set to 0x2020. After that, addition, subtraction, and multiplication are performed on r31 to make its value 0x18. Then the “st” command is used to store the value of r31 into the X register in SRAM.

After the SUB instruction, which in one case yields a negative value, the status register bits only store the last bits of the answer since SUB drops overflow bits. BRMI checks to see if the Negative flag is activated in the status register. In this case, it finds that it is activated, so it jumps to the “here:” line, skipping the next three lines of code.

Several more commands are used in this program. Here are their meanings, as well as how they are used in this scenario:

* LSL stands for “logical shift left”, meaning it shifts all bits left by one place, effectively multiplying the number by 2. In this program, LSL multiplies number in r30 by 2, which is AC \* 2 = 0x158, but only 0x58 can be stored and the overflow flag is activated.
* 
* LSR stands for “logical shift right”, meaning it shifts all bits right by one place, effectively diving the number by 2. In this program, LSR divides the number in r30 by 2. 0x58 divided by 2 = 0x2C.
* 
* ASR stands for “algorithmic shift right”, which has the same functionality as LSR except it won’t change the most significant bit that affects the number’s sign (e.g. positive or negative). This again divides the number in r30 by 2, 0x2C / 2 = 0x16.
* BSET sets the bit indicated to 1 (i.e. activates the bit). In this case, “bset 2” activates the second bit in the status register, which is the negative flag. This makes the r30 value negative 0x16.
* 
* BCLR sets the indicated bit to 0. In this case, “bclr 2” deactivates the negative flag in the status register. The value in r30 is now set back to positive 0x16.
* 
* Because the value in r30 is positive, “brmi here” evaluates to false and the program continues to the next line of code rather than jumping to “here”.
* In SRAM location 0x2020, the value 0x16 is stored. This happed after the “st X+, r30” instruction, which stores the value in r30 (0x16) into the SRAM location referenced by X (0x2020), then increments X by 1.
* By the same process, SRAM locations 0x2021 and 0x2022 are stored with the value 0x16 by the subseqent commands (which are also “st X+, r30”). Here is evidence of that: 

Source Code (Software)

/\*

\* project1.asm

\*

\* Author: Eugene

\*/

//Bit Instructions

.cseg ;FLASH code segment

.org 0x0

start: ldi r26,0x20

ldi r27,0x20

ldi r30,0xAC

lsl r30 ; logical shift left; multiplies number in r30 by 2 (AC \* 2 = 0x158, but only 0x58 is stored + overflow is activated)

lsr r30 ; divides by 2 (in this case, r30 just has 0x58 bc overflow; dividing by 2)

asr r30 ; algorithmic shift right; never change the original sign of the number; divides by 2 ; it's unsigned, so it keeps the first bit as a not a sign digit

bset 2 ;activates negative flag (in bit 2)

bclr 2 ;sets neg flag to 0 (in bit 2)

brmi here

st X+,r30 ;stores the value of r30 in address pointed to by X, then increments the address by 1 (moves to next address)

st X+,r30

st X+,r30

here: jmp here

.exit

/\*

.eseg ;EEPROM segment

.org 0x1000

eevar: .dw 0xfeff

msg: .db "Hello World", '\n', 0

.dseg ;SRAM data segment

.org 0x2000

string: .byte 3 ;a three byte variable/\*

/\*

\* project1.asm

\*

\* Created: Septmber 5, 2017

\* Author: Zachary Shumate

\*/

/\*.cseg ;FLASH code segment

.org 0x0

start: ldi r30,56

ldi r31,24

add r31,r30

here: jmp here

.exit

.eseg ;EEPROM segment

.org 0x1000

eevar: .dw 0xfaff

msg: .db "HelloWorld",'\n',0

.dseg ;SRAM data segment

.org 0x2000

string: .byte 3 ;a three byte variable

\*/

; Part 3

/\*

\* project1.asm

\*

\* Created: (date)

\* Author: (Name)

\*/

//Arithmetic, logic, Data Transfer Instructions

/\*.cseg ;FLASH code segment

.org 0x0

start: ldi r26,0x20

ldi r27,0x20

ldi r30,56

ldi r31,24

add r31,r30

sub r31,r30

and r30,r31

mul r30,r31

st X,r30

clr r30

ser r31

here: jmp here

.exit

\*/

/\*

.eseg ;EEPROM segment

.org 0x1000

eevar: .dw 0xfeff

msg: .db "Hello World", '\n', 0

.dseg ;SRAM data segment

.org 0x2000

string: .byte 3 ;a three byte variable

\*/

/\*

\* project1.asm

\*

\* Created: (date)

\* Author: Eugene

\*/

//Conditional Branch instructions

/\*.cseg ;FLASH code segment

.org 0x0

start: ldi r26,0x20

ldi r27,0x20

ldi r30,56

ldi r31,24

sub r31,r30

brmi here

st X,r30

nop

nop

here: jmp here

.exit

\*/

/\*

.eseg ;EEPROM segment

.org 0x1000

eevar: .dw 0xfeff

msg: .db "Hello World", '\n', 0

.dseg ;SRAM data segment

.org 0x2000

string: .byte 3 ;a three byte variable

Schematics (Hardware)

None.

Application-Based Analysis

Knowing the basics of Atmel Studio and Assembly programming can be applied to many real-world solutions using embedded systems. These may include: traffic lights, automobiles, sensors, appliances, digital cameras, calculators, manufacturing robots, and more.

Conclusion

The purpose of this project was to learn the basics of Assembly programming with Atmel Studio 7. Based on the fact that I’m now comfortable using the software to step through and understand Assembly code, this goal was achieved. I was able to analyze changes in the register and SRAM values as I stepped through the program to understand what each instruction did.

References

I used the Atmel web pages titled “AVR Assembler Instructions” and “AVR Assembler Directives” (found here: <http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_directives.html>).