# **Decisions, Decisions, Decisions**

**Introduction:**

The purpose of this lab is to get experience with how branch commands are used to make decisions as well as how the stack frame is setup to hold the return address and crucial data from functions or the status register. We will also continue working with the assembly language so you can see how to access the different internal resources that include:

* General Purpose Registers (GPRs) that are inside the AVR Core
* Special Function Registers (SFRs)
* SRAM (8 bits wide)
* Flash (16 bits wide)
* Bootloader Area of Flash
* EEPROM

**Note: \* The last two areas of memory on this list we will cover in later material**

The demo code did not use any decision structure for the delay loop. What it did was check to see if a button was pressed and then jumped down sequentially to the delay loop before returning to the start of the code where the LEDs were updated with whatever button was active (e.g. rotate, increment, etc.). There are many

## **Outcomes:**

1. Code AVR Assembly language instructions to create loops.
2. Code AVR Assembly language conditional branch instructions.
3. Code AVR programs to generate a time delay (Demo Code revisit)
4. Calculate target addresses for conditional branch instructions.
5. Describe the stack frame and its use in subroutines and functions.
6. Have a working knowledge of pipelining in the AVR.
7. Have a working knowledge of crystal frequency versus instruction cycle time in the AVR.

## **Parts and Equipment:**

1 - Development Board and "Starter Kit" components

Atmega32 Datasheet

Microchip Studio IDE

Atom Text Editor

**Procedure: Warm-up with More Basic Assembly Code**

**Part I**

In general terms there are four major types of addressing modes used for any type of architecture. They are 1) Inherent Addressing Mode; 2) Direct Addressing Mode; 3) Immediate Addressing Mode; and 4) Indirect Addressing Mode.

**Inherent Addressing commands have one operand (e.g., inc r16; dec r17; ror r20 etc.)** Before this command can be executed the general purpose, register needs to have a value loaded into it. The direct addressing mode in the AVR can include between GPRs and between GPRs and SFRs where we are sending a value out to a port location. **Typical direct addressing mode examples is when we are adding, subtracting, multiplying, or dividing values between two GPRs. (e.g., add r16, r17; sub r20, r21; mul r3, r4). Notice with these commands there are two operands.**

An immediate addressing mode op-code allows us to load into a GPR a piece of data directly. The data can be in binary, octal, decimal, or hexadecimal. **We can also use assembly directives such as .set and .equ to give a label to the value to make it a constant. In addition, we can use another assembly directive (.def temp = r16) to provide a username to a GPR to make our code more readable and manageable.** Examples of immediate op-codes are: **ldi temp, $FF; cbi portb, 0;**. The last addressing mode is indirect. **Indirect addressing mode is used with the X, Y, and Z registers (GPRs R26:R27, R28:R29, R31:R30)** to point to locations “indirectly”. Yes, for those C++ students these are pointers. **We place a base address and offset value into the 16-bit register location to indirectly point to a location and then increment the offset value to get to the next location. Indirect addressing is extremely helpful in setting up “Look-Up Tables” in any memory location.**

**The commands IN, OUT, STS, and LDS are examples of direct addressing mode between the GPRs and SFRs or GPRs and SRAM.** Notice that the SFR registers start at address 0x20 in SRAM or address 0x00 in an extended I/O address location. The SRAM address is in parentheses when looking the register summary in the datasheet. **For example, DDRB has a location 0x17 (0x37).** If you wanted to send a value out to DDRB:

* OUT DDRB, r16 ; uses the I/O label with address in the include file
* STS 0x37, r16 ; stores value at memory location

Either command is acceptable. The first one is easier to understand and is more universal between different devices. Most AVR devices have a port called B and using the label with the include file pointing to the location means you do not need to track down the different memory locations as you switch devices. If you were creating a bare-metal project you would create a header file (reg32.h for example) that is like the default include file (m32def.inc) that contains all the addresses of these registers.

1. Write and assemble a program to load register R20 with value $99. Then from register R20 move it to R0, R12, and R31. Use the simulator to single-step the program and examine the registers. Check the program counter (PC) register as you single step through each opcode. Document in your comments what each statement is doing and notes on the results of this code.
2. Write and assemble a program to add all the single digits of your ID number and save the result in R16. Pick 7 random numbers (all single digit) if you do not want to use your ID number. You may choose for extra points to use assembly directives to create constants (.equ versus .set). Then use the simulator to single-step the program and examine the registers. Check both the PC and SREG registers. The SREG will track if flag bits for carry, half carry, etc. Document in your comments what each statement is doing and notes on the results of this code.

### **Procedure:**

### **Part II – Branch Instructions and Looping**

In your previous experiences you have used if-else; do-while; while; and for loop are examples to repeat a set of instructions and/or to check for a particular set of conditions before a task is completed. In this section we will take a look at some of the more common branch instructions. You have already been exposed to one in the Demo Code that used BRNE or branch if not equal in a nested loop to slow the LEDs down so we can see the changing patterns on the LED bar.

1. In example 3-1 in the textbook simulate the program code in Microchip Studio. Take a screen shot of the GPR registers and PORTB in the simulator when the final sum is determined, and the value is sent out to PORTB. Document in your comments what each statement is doing and notes on the results of this code.
2. In example 3-3 in the textbook simulate the program code in Microchip Studio. Change the size of the inner and outer loop to 3 and 5 (10 and 70 is a bit much to single step through) and observe the code running. Take a screen shot of PORTB changing.

Note: In both of the examples above the author left out the initialization of PORTB which is done in the register DDRB (data direction register b). Placing one’s in the register configure the port for output and zero’s make it an input:

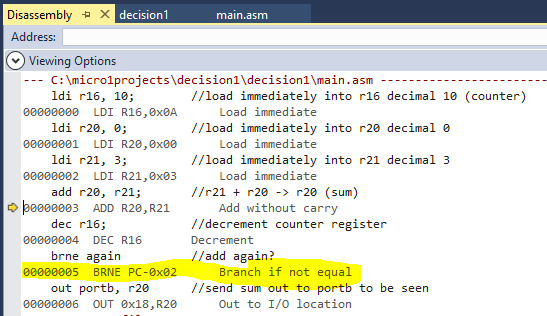
LDI R16, $FF ;load immediately all ones into R16

OUT DDRB, R16 ;copy R16 value into DDRB

If you decide to take Example 3-3 out to the development board for extra points increase the size of the loop by making the values large.

1. Complete one of the other branch examples of your choice and Document in your comments what each statement is doing and notes on the results of this code.

Besides seeing the values change in the GRPs, SFRs, and SRAM we also want to track what the program counter (PC) is doing and understanding how the branch and jump commands work. Branch commands have a limitation on how far they can jump. There limit is 64 bytes of the PC. If the instruction is past the range there will be an out of range error and the code will not compile. To do this we will go back and look at example 3-1. Remember as you complete the number of addresses jumped that the program counter is always one step ahead of you. The process for determining the range is straightforward. When jumping forward the value is positive; but when jumping backwards the value is negative (2’s complement value).



**Figure 1: Disassembly Window for Example 3-1**

The steps to calculating the displacement value for the branch address are:

* Open the Disassembly window in Simulation mode of the project. Your window will be the same as Figure 1.
* The values in “gray” represent the address of the opcode in flash memory as well as the opcode, operand(s).
* Find the address location of the branch command and the address of where it will jump to. In the case of BRNE it will jump if not equal (This statement uses the Z bit in the Status Register).
* Calculate the displacement (how many addresses forward or backwards).

1. Calculate the displacement for Example 3-1. Show all of your work including converting into the 2’s complement value.

0111

+1001

10100

Note: The RJMP, RCALL, and IJMP commands also have limitations to how far they can jump. The process is the same for these commands it is just the distance is a bit larger in size.

### **Procedure:**

### **Part III – Call Instructions and Stack Frame**

The focus on this section of the lab will be how to setup the stack so we can use it when we develop subroutines and functions. Not all instructors make a difference between the two but I do. To me a subroutine is a gadfly loop that does not do anything – like a delay loop. Whereas a function or method does a task such as turning on a motor based on a sensor input, retrieving data from memory, etc.

**There are three opcodes used for subroutines or functions: 1) CALL, 2) RCALL, and 3) ICALL**; the only difference between the first two is the range of addresses. The third one is used with interrupts. **All CALL commands have a companion that goes at the end of the subroutine or function called return (RET or IRET).** The return statement needs to be there otherwise you will not be able to go back to the next command after the call command.

**The call command pushes unto the stack the return address (PC is +1 ahead looking at the next command). The return command pops off the stack the return or Old PC address after the function has been executed.** There are two other commands that are used with the stack frame: 1) PUSH and 2) POP. **These two commands allow us to push the GPR data unto the stack or pop the data back into the GPR.** The key when using these commands is to do a balanced push and pop from the stack; so if you push three items unto the stack make sure you pop three items off the stack in reverse order. Otherwise, you will bury the Old PC address on the stack and jump back into the Twilight Zone…due..do…due…do!

To use the stack we must first set it up. There is a register that keeps track of the location of the stack frame top and decrements the location when data is pushed unto the stack. When data is removed from the stack this register increments back up to the top of the stack. **This register is called the Stack Pointer Register (SP) and is broken into two 8-bit registers (SPH:SPL) for high and low byte. You can place the Stack Top Address using the load immediately and out commands into any available RAM location (0x0060 through RAMEND).** However, if you choose an address other than RAMEND be careful about how many pieces of data that you plan on pushing unto the stack. Remember that the GPR registers are 0x0000 to 0x001F of SRAM and you do not want to write-over the top of data in those registers. To use RAMEND as the address location the four commands below setup the stack at RAMEND which is away from the GPRs. These commands are incorporated into the initialization section of any project before reaching the main project code.

**LDI R16, HIGH(RAMEND) ; load the high byte of RAMEND into R16**

**OUT SPH, R16 ; send value out to stack pointer high byte register**

**LDI R16, LOW(RAMEND) ;load the low byte of RAMEND into R16**

**OUT SPL, R16 ;send value out to stack pointer low byte register**

Note: We cannot load a value directly into a SFR location. We use the GPRs to load the value and then use the OUT command to “copy” the value into the SFR.

The nice thing about using the label RAMEND is that all include files have a label called RAMEND with the physical address tied to it. So, you as a programmer do not need to track that address for each device used. Writing bare-metal code you will create your own label name for this location in your header file.

1. Create a project for example 3-10 and document the addresses in the stack frame after each call by taking a screenshot during simulation. Reduce the size of the delay loop at address 0x300 to 5 to simulate your code.

Extra Credit: Add the initialization code PORTB and increase your delay loop so you can see the results. If need be use the demo code delay loop in place of the single register delay loop. Document your code.

1. Write and assemble a program to:

Set SP = 0x049D,

Put a different value in each of RAM locations $049D, $049C, $049B, $049A, $0499, and $0498,

POP each stack location into registers R20 – R24.

Use the simulator to single-step and examine the registers, the stack, and the stack pointer.

Take a screen shot of your results.

#### Submission Details:

Ck Lab – Submit this document with typed responses to the questions below. Upload it to the assignment.

1. Completed Lab Work and include the following questions in your write-up.
   1. What commands are used to setup the stack frame?
   2. When pushing items onto the stack does the stack pointer register increment or decrement? When popping items off the stack?
   3. In your own words describe what a branch penalty is and why it is important to consider it?
   4. Upon reset the SP points to location \_\_\_\_\_\_\_\_\_\_\_\_.
   5. What questions do you still have about simulating code within Microchip Studio?
   6. Other questions in the lab?

Lab Synopsis –

1. Create a folder called Lab3YourName
2. Copy your cleaned projects into this folder.
   1. Simulated Work with comments for full credit
   2. Screen Shots