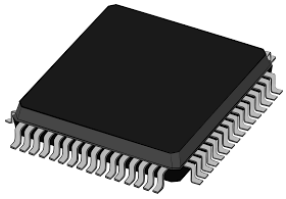
Computer Organization

Lab 3



EITF70

Low Level Programming

Goals

- Get an understanding of how a CPU works
- Understanding the basics of assembly programming
- Be able to mix C and assembly code
- Understanding when and why assembly language should be used

Sidra Muneer, Steffen Malkowsky and Erik Larsson
2020
Christoffer Cederberg and Jonathan Sönnerup
2019

Contents

Introduction

Purpose

The purpose of this laboratory exercise is to get a basic understanding of assembly programming, try to mix a program with C and assembly code, get familiar with memory management, like the stack and calling conventions, that is passing parameters to functions and subroutines.

Organization

The material is organized in three main parts; assembly programming, memory management subroutines, and calling conventions - mixing C and assembly. For each part, there is first an introduction which mainly aims at describing the specifics of AVR and, second, there is laboratory exercises, detailing tasks, home assignments and laboratory questions.

Assembly Programming

In this chapter we focus on Assembly programming. Just like any programming language, we have a certain amount of instructions available in assembly to form a program.

The chapter is organized as follows. First, we briefly introduce Assembly programs, conditional expressions and input/output (I/O). We then discuss how to computer and measuring execution time and finally the lab exercises are detailed.

A Simple Program

A simple but working example in AVR is shown in Listing 1. For a listing of useful instructions available in the AVR processor, see Appendix A.

```
start:
    ldi r16, 12     ; load 12 into register 16
    ldi r17, 13     ; load 13 into register 17
    add r16, r17    ; add r16 and r17, save result in r16
    cpi r16, 26     ; compare r16 with the values 26
    breq end        ; jump to end if true
    add r16, 1      ; add 1 to r16, r16 now holds the value 26
                    ; after this instruction, we will execute the rjmp end
end:
    rjmp end        ; infinite loop
```

Listing 1: Simple AVR assembly program.

As seen in the code, there is no `main` function. In AVR assembly, the name of the main function is `start`. The names `start` and `end` are known as *labels*. This is just a name of the actual address where the code resides, to make it easy for a programmer. A label can be used in two ways: to create a place to jump to, like a for-loop or similar, but also to create a function (subroutine) where the label must be called with the `call` instruction. The `end` label in Listing 1, together with the "`rjmp end`" instruction, is used to create an infinite loop, just like a `while(1)` loop in C.

Conditional Expressions

Operations performed in the ALU (arithmetic logic unit) have typically an affect a so called status register, `SREG`. The status register contains flags such as overflow, carry, and zero. For example, the zero flag, `Z`, is set to 1 if the result after an operation is 0. The advantage with storing these flags from an ALU operation is that

this information can be used by later instructions to create conditional expressions, such as if statements. In Listing 1, we have such a case. The `cpi` instruction compares the value of a register, `r16`, with the value 26. The compare operation is actually a subtraction in the ALU. The result is either 0 or not, and the `Z` flag in the status register is set accordingly. Next, the `breq` instruction (branch if equal) automatically checks the `Z` flag to decide whether to jump (branch) or not. In this case, the result from the `cpi` instruction is $25 - 26 = -1$, and the `Z` flag is set to 0 (the result was not zero). Hence, the branch is not taken, and the program continues with the next instruction, in this case, an `add` instruction.

Input/output (I/O)

To communicate with the surrounding circuits, we need to access the I/O registers. For that, we use the `in` and `out` instructions. An example is shown in Listing 2. Note that just as in C, we can define names to numbers and addresses to make it easier to remember. In this case, we defined `DDRB` and `PORTB` to avoid writing the addresses in the assembly code. To make it more interesting (and a little confusing), there are basically two ways of accessing I/O ports in assembly. Either by using normal load and store instructions, or by using special I/O instruction such as `in` and `out`. There are 2 differences, one being that the I/O instructions are more efficient (1 clock cycle instead of 2), the other being that the value $0x20$ should be subtracted from the addresses to the I/O ports. The reason is that they are mapped in different ways in hardware. Table 1 shows the addresses when using the special I/O instructions. **Note** that using `in` and `out` is preferable to use.

Table 1: I/O registers and their corresponding addresses.

Register	**Address**
PINA	0x00
DDRA	0x01
PORTA	0x02
PINB	0x03
DDRB	0x04
PORTB	0x05
PINC	0x06
DDRC	0x07
PORTC	0x08
PIND	0x09
DDRD	0x0A
PORTD	0x0B

Measuring Execution Time

Every instruction takes a certain amount of clock cycles. We can use that information in order to evaluate the execution time of a program. Some instructions takes different amount of clock cycles based on the outcome. For example, due to the pipeline a branch instruction takes 1 clock cycle if the branch is **not** taken, and 2 clock cycles if it is. Analyzing the program in Listing 3 yields the result shown in Table 2. Note that in order to analyze the program, we must unroll the loop, counting clock cycles based on if the branch is taken or not. The total execution time (before the infinite loop) is 9 clock cycles. Running at 16 MHz,

```
#define DDRB    0x04
#define PORTB   0x05
#define LED0    0

start:
    ldi r18, (1 << LED0)  ; |
    in  r17, DDRB         ; |
    or  r17, r18          ; |
    out DDRB, r18         ; V DDRB |= (1 << LED0);
    in  r17, PORTB        ; |
    or  r17, r18          ; |
    out PORTB, r18        ; V PORTB |= (1 << LED0);
end:
    rjmp end        ; infinite loop
```

Listing 2: Simple AVR assembly program using I/O.

this yields

$$9 \cdot \frac{1}{16\text{ M}} = 562.5\text{ ns}.$$

```
start:
    ldi r16, 1          ; load 1 into register 16
    ldi r17, 13         ; load 13 into register 17
    loop:
        add r17, r16    ; add r17 and r16, save result in r17 (r17++)
        cpi r17, 15     ; compare r17 with the values 15
        brne loop       ; jump to loop if r17 != 15
end:
    rjmp end            ; infinite loop
```

Listing 3: Simple AVR assembly program.

The general formula for calculating the **average** execution time of a **single round** in a loop (say, a for-loop) is

$$\frac{n-1}{n} \cdot p + \frac{1}{n} \cdot q, \tag{1}$$

where n is the number of rounds, p is the number of clock cycles in a round when the branch is taken, and q is the number of clock cycles during the last round (branch not taken). Using the formula for the loop in Listing 3, we get

$$\frac{2-1}{2} \cdot (1+1+2) + \frac{1}{2} \cdot (1+1+1) = \frac{1}{2} \cdot 4 + \frac{1}{2} \cdot 3 = 3.5.$$

That is, the average execution time is 3.5 clock cycles. Running for 2 rounds yields 7 clock cycles, which matches the number in Table 2. For very long loops, the last round does not affect the average time very much. For example, running the same loop 100 times instead would yield

$$\frac{99}{100} \cdot 4 + \frac{1}{100} \cdot 3 = 3.99,$$

which is basically 4 clock cycles per round, which we get if do not take into account that the last branch instruction only takes 1 clock cycle. In general, for large numbers, n, we get

$$\lim_{n\to\infty} \frac{n-1}{n} \cdot p + \frac{1}{n} \cdot q = p.$$

Table 2: Caption

Instruction	Execution time
ldi r16, 1	1
ldi r17, 13	1
loop:	
add r17, r16	1
cpi r17, 15	1
brne loop	2 (taken)
———	
add r17, r16	1
cpi r17, 15	1
brne loop	1 (not taken)
———	
end:	
rjmp end	∞

What constitutes as a *large* number depends on the application. Make sure not to approximate too early, or you may end up with a completely wrong estimation.

Lab Exercises

Just as always when we are to try new things, we implement a simple program as a sanity check. Here, we will use the AVR assembly language to blink an LED.

Tasks:

- Create a new project in Atmel Studio, choose "AVR Assembler", under the "Assembler" tab to the left, instead of "C/C++".

> **Home Assignment 1**
> How many clock cycles are needed if a delay of 0.1 s is desired with a clock frequency of 16 MHz?

> **Home Assignment 2**
> How many bits do we need to count to the value above?

> **Home Assignment 3**
> How many registers do we need for the delay of 0.1 s?

> **Home Assignment 4**
> Write a snippet of assembly code that delays the program for roughly 0.1 s. Remember that the jump instructions also take time to execute.

> **Home Assignment 5**
> How do you turn on LED 2 on port B in assembly? Remember to set the direction to an output. Refer to Appendix A for instructions.

- Use your delay code to create a program that blinks an LED at 5 Hz.

> **Lab Question 1**
> How can you use your delay code in order to create arbitrary delays? What is the limitation?

- Make the LED blink at 1 Hz instead.

You are now done with this part, show your work to a lab assistant! _____ ☑

Memory Management and Subroutines

This chapter covers the primary memory, also known as the memory or the random access memory (RAM). In particular, the chapter covers organization, access and operations on the memory, the stack and subroutines. The chapter is organized as follows. First there is theory on how to read and write data in the memory, stack and subroutines. Second, the chapter contains laboratory exercises.

Accessing the Memory

The AVR processor's work registers are only eight bits long, which means that the largest integer that fit in them are 255. The RAM of the microcontroller is 16 kB. How is it then possible to use the entire memory space? Of course, the designers of the processor had a solution. A couple of the work register can be used as one 16 bit register, namely r26 and r27. The same is true for r28-r29 and r30-r31. The register pairs are named X, Y and Z. See Figure 1 below.

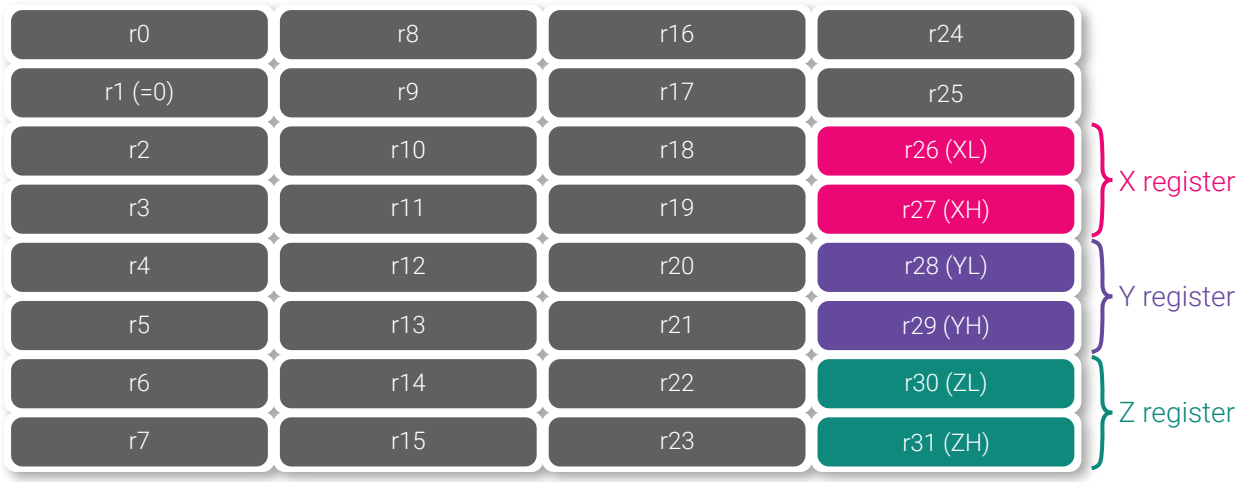


Figure 1: The register file in the processor with the X, Y and Z register highlighted.

To access the register one can use the macros in Table 3.

In Listing 4 an example on how to use the Z register can be seen.

Table 3: Macros for the X, Y and Z register.

Macro	Description
X	r26:r27
XL	r26
XH	r27
Y	r28:r29
YL	r28
YH	r29
Z	r30:31
ZL	r30
ZH	r31

```
#define PINA    0x00

start:
    in      r22,    PINA        ; Loads r22 with PINA
    ldi     ZH,     0x40        ; Sets low byte of Z reg to low byte of SP
    ldi     ZL,     0xF0        ; Sets high byte of Z reg to low byte of SP
    st      Z,      r22         ; Store r22 to addres in Z register
    ld      r21,    Z           ; Load r21 with RAM content at address Z
    rjmp    start
```

Listing 4: Accessing data in RAM.

The Stack

The stack is the part of the RAM used for local, temporary, variables. In C, the allocation of local variables is handled automatically by the compiler. In assembly, this must be done by you. The stack grows from higher addresses to lower. This means that when allocating memory on the stack, we must **subtract** the stack pointer with number of bytes we want allocated. When returning the memory, we **add** the same number to the stack pointer. Failing to properly return stack memory will result in program crashes. Recall that when calling subroutines, the return address is being stored on the stack, hence failing to manage the stack makes a subroutine return back to the wrong address.

A stack pointer is used to keep track of the "top". For an AVR processor, the stack pointer is a 16-bit, memory mapped, value located in the memory at address 0x3D (no offset). On start-up, the stack pointer is set to the end of RAM, i.e., the highest address, which is 0x40FF. For each byte added to the stack the stack pointer is decremented. This means that the stack is growing towards lower addresses.

There are two special instructions directly associated with the stack. The first is `push`. With this instruction the content of the specified register will be stored at the address given by the stack pointer. After a `push` instruction is performed the stack pointer is decremented (the stack grows towards lower addresses). The second instruction is the `pop`, which is the opposite. It move the last added byte from the stack to the specified register and increment the stack pointer.

Another way of allocating and deallocating space on the stack is to add or subtrack a given number from to/from the stack pointer. An example of this is shown in Listing 5. The code allocates two bytes on the stack and then stores two bytes of data. This is achieved by:

- loading the stack pointer into `r28` and `r29` (later used as the Y register),

- subtract two to `r28` and `r29` (the result is stored in the same registers),
- writing the content of `r28` and `r29` back to the stack pointer register,
- loading `r20` and `r21` with data from address `0x63` and `0x59`, respectively,
- writing `r20` and `r21` to the stack, using the Y register, i.e., `r28` and `r29`.

```
#define STACK_H 0x3E    ; Address to the high byte of the stack pointer
#define STACK_L 0x3D    ; Address to the low byte of the stack pointer
#define N_ALLOC 2

warp:
    .
    .                   ; Code to initialize anti-annihilation chamber
    .
    in r28, STACK_L     ; Load low byte of stack pointer to r28
    in r29, STACK_H     ; Load high byte of stack pointer to r29
    sbiw Y, N_ALLOC     ; Subtract N from the loaded stack pointer
    out STACK_L, r28    ;
    out STACK_H, r29    ; Update stack pointer
    ldi r20, 0xCC       ; Load r20 with warp core start sequence
    ldi r21, 0xA1       ; Load r21 with wormhole coordinate system variable
    std Y+1, r20        ; Store content of r20 on the stack
    std Y+2, r21        ; Store content of r21 on the stack
    .
    .                   ; Warp core calibration code
    .
    ret
```

Listing 5: A snippet of assembly code that allocates an array with the size of 2 bytes on the stack.

The stack before and after running the snippet of code can be viewed in Figure 2. As seen, the stack pointer has been decremented after the execution and the two bytes of data has been added. It should be noted that the stack pointer always points on the next free byte, which is below (lower address) the "top of stack"[1].

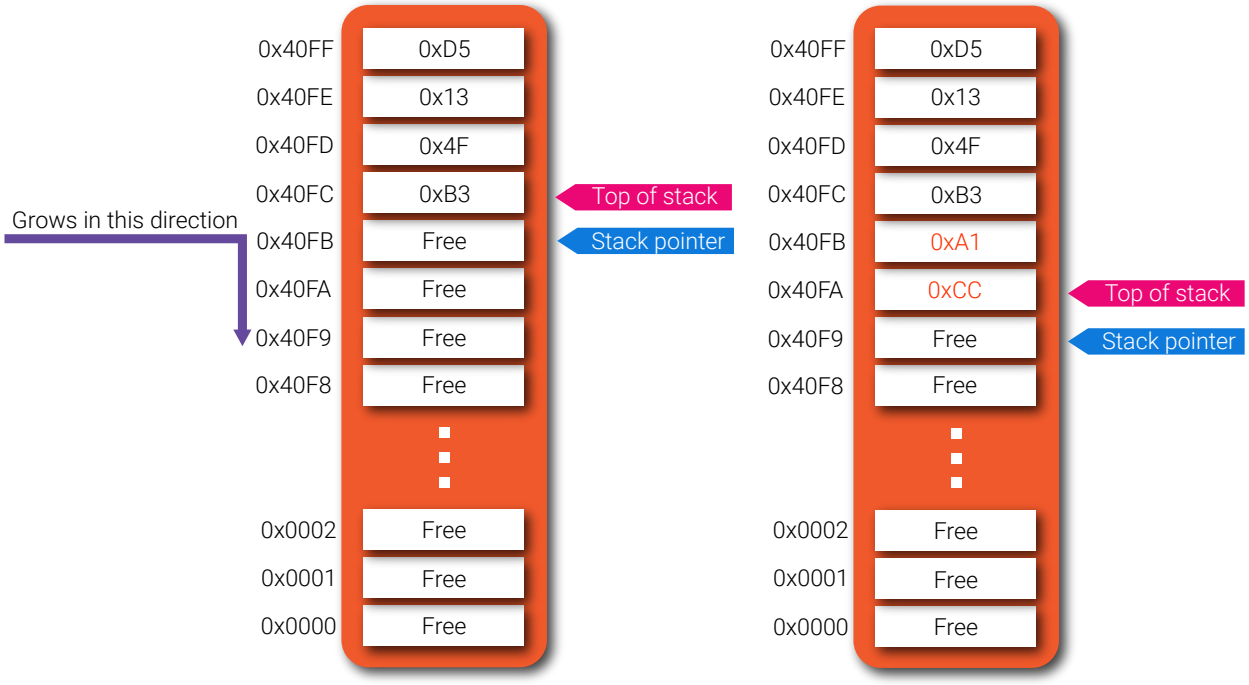


Figure 2: The stack before (left) and after running the code in Listing 5.

Whenever the stack has been used, that is, the need for storing data is no longer present, the allocated bytes need to be deallocated. If this is not done, the stack will keep growing and soon or later it will collide with the other content stored in the RAM. To return the allocated space, the number of bytes that was allocated should be added to the stack pointer. By doing so the memory will be freed up, see Figure 3. Do note that the content of the previously used memory space is still there (but it should be considered to be lost).

[1]The top of stack is purely imaginary, there is no register in the CPU that contains this address.

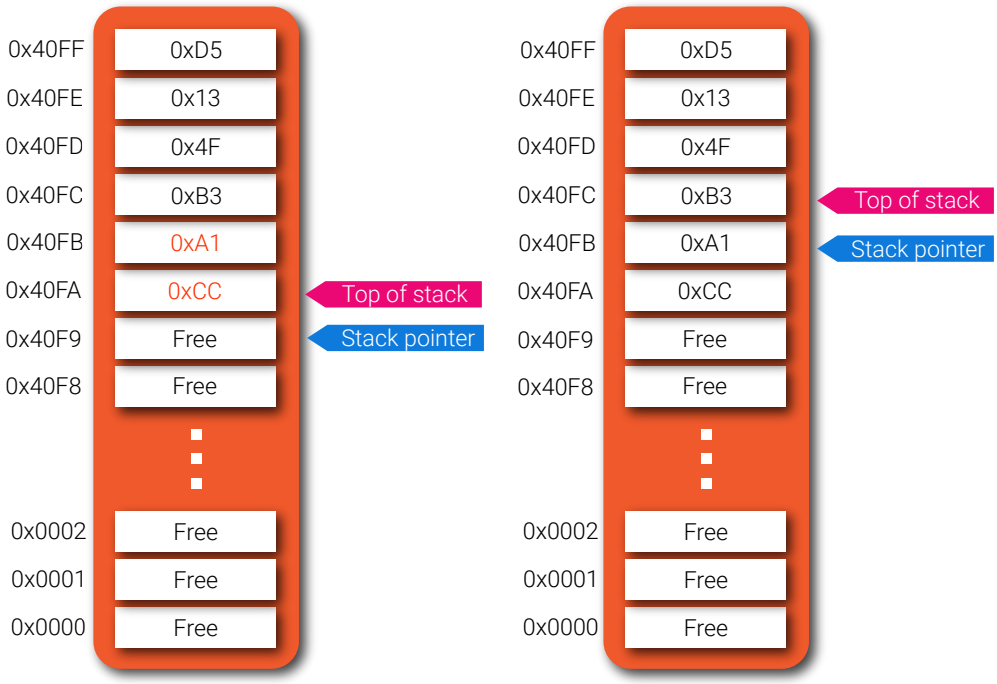


Figure 3: The stack before and after (right) the allocated bytes have been "returned".

Subroutines

Just as we can separate code in different functions in C and Java, assembly lets us separate code in different subroutines, using labels. As before, a label can be used both in loops to jump to, but also to declare a subroutine. The difference is how they are used. To jump to a label (say, in a loop), any branch instruction may be used. When declaring a subroutine, it must be called using the `call` instruction.

When calling a subroutine, using `call`, an important thing happens in the processor. The address of the instruction after the call, which is PC[2] + 1 (the digit 2 is a footnote), is pushed on the stack. This is important since we must be able to continue where we left off. An example is shown in Listing 6.

```
start:
    call delay          ; delay forever
    rjmp start

delay:
    ldi r16, 100        ; r16 = 100
    loop:
        dec r16         ; r16--
        brne loop       ; loop if r16 != 0
    ret                 ; return from subroutine
```

Listing 6: Calling subroutines in assembly.

As shown in the code, we can use labels in both ways at the same time, here used both for declaring a subroutine, `delay`, and also for the loop. Every subroutine must end with a `ret` instruction. This will pop the return address from the stack and jump back to the `start` routine by setting the program counter to the popped value. After this, `delay` will be called again, and again.

For the keen reader, a disassembled[3] version of the code can be found in Listing 7. To make life a little bit harder than it already is, it needs to be said that the program memory (the FLASH) is word-addressed, where one word is 16 bits, in contrast to the RAM, which is byte-addressed. Furthermore, some instructions, for instance the `call` instruction, has the length of two words (32-bit), whereas many others only are one word (16 bits) long, thereof the gap in addresses between the `call` and `rjmp` instruction in Listing 7.

```
;Address    Opcode (disassembled code)  Assembly Instruction    Comment
;-------------------------------------------------------------------------------------
                                       start:
00000000    CALL 0x00000003            call delay              ; Calling the delay subroutine
00000002    RJMP PC-0x0002             rjmp start              ; Jump back to start
00000003    LDI R16,0x64               ldi r16, 100            ; Load r16 with 100
00000004    DEC R16                    dec r16                 ; Decrement r16 with 1
                                       loop;
00000005    BRNE PC-0x01               brne loop               ; Branch if not equal to zero
00000006    RET                        ret                     ; Return to address 00000002
```

Listing 7: The dissasembled code from Listing 7.

[2]This is the program counter which contains the address to the instruction that is executed at the moment.

[3]A disassembled code is the true code that the processor will execute, i.e., code without labels, `#define`s and so on. The assembly code can not in all cases be direct mapped to raw machine code.

The Anatomy of a Subroutine

A subroutine is a bit more primitive than its high level counterpart in the sense that the programmer needs to do some parts of the compiler's job. What the programmer needs to do will be addressed in this section.

A subroutine is divided into three parts; a prologue, a body, and an epilogue, see Figure 4. Each part has its specific purpose.

Figure 4: The anatomy of a subroutine.

When a subroutine has been called, it is important the every register, that will be used during the call and which content is of importance to the caller after the subroutine has executed, is being saved on the stack. This should, of course, be done in the beginning of the subroutine. The registers are preferably saved with the `push` instruction. Furthermore, if more space on the stack is required, it should also be allocated here (for ease of use, after the pushing the registers). This part of the code is referred to as the prologue. In Listing 8, an example of this is shown. In the subroutine there is a need to save `r28` and `r29`, but not `r19`.

```
#define STACK_H 0x3E     ; Address to the high byte of the stack pointer
#define STACK_L 0x3D     ; Address to the low byte of the stack pointer
#define N_ALLOC 5

defense_routine:
    ;==================== Start of the prologue
    push r28
    push r29
    in r28, STACK_L      ; Load low byte of stack pointer to r28
    in r29, STACK_H      ; Load high byte of stack pointer to r29
    sbiw Y, N_ALLOC      ; Subtract N_ALLOC from the loaded stack pointer
    out STACK_L, r28     ;
    out STACK_H, r29     ; Update stack pointer
    ;==================== End of the prologue


    ;==================== Beginning of subroutine body
    .
    .
    .
    .
```

Listing 8: The prologue of an assembly subroutine.

In the part of the routine called the subroutine body, the routine performs its designated task. When this

is done, the epilogue starts. The first thing to do here is to free up all memory that has been allocated on the stack. The second thing is to restore the registers that were pushed on the stack. This is achieved by adding an appropriate value to the stack pointer. When this is done, the stack pointer will be pointing at the last value that was pushed to the stack in the prologue. When all of this is done, the subroutine can return back from where the subroutine was called (with the `ret` instruction). In Listing 9, an example of an epilogue is shown.

```
#define STACK_H 0x3E    ; Address to the high byte of the stack pointer
#define STACK_L 0x3D    ; Address to the low byte of the stack pointer
#define N_ALLOC 5

defense_routine:
    .
    .
    .
    ;==================== End of the subroutine body

    ;==================== Beginning of the epilogue
    in r28, STACK_L     ; Load low byte of stack pointer to r28
    in r29, STACK_H     ; Load high byte of stack pointer to r29
    adiw Y, N_ALLOC ; Add N_ALLOC to the loaded stack pointer
    out STACK_L, r28    ;
    out STACK_H, r29    ; Update stack pointer
    pop r29
    pop r28
    ret
    ;==================== End of the epilogue and the subroutine
```

Listing 9: The epilogue of an assembly subroutine.

Lab Exercises

Dynamic Memory Allocation

Sometimes, the general purpose working registers are not enough to store the data you use. Assume that you are sampling your beautiful singing voice from a microphone over a time period. With only 32 registers in the AVR, we will run out of space pretty quick. In such a case the RAM is utilized. For global variables the `.data` section is used. For local variables (inside a function) the data is usually stored on the stack.

The authors recognize that this assignment is a bit artificial, but it highlights some interesting points students need to know, in a simple way.

Tasks:

- Create a new "Assembler" project in Atmel Studio.

  Home Assignment 6
  How do you allocate 10 integers on the stack?

- Write an assembly subroutine that allocates an array of 5 bytes on the stack.

  Lab Question 2
  What is the value of the stack pointer directly after allocation? Use the debugger.

  Home Assignment 7
  If you allocate an array of bytes on the stack, and the stack pointer after allocation is 0x40E0, what is the address of the value at index 3?

- In the subroutine, sample the state of the buttons B1 to B6 in a loop and save the five latest in the array. The first sample shall be placed at index zero (arr[0]), the fifth sample at index four. After saving the values, simply return from the subroutine.

  Remember that when you borrow something, you must return it.

- In the main function, call your subroutine in a forever-loop.
- Start the debugger and step through each sample to verify that you can read the values of the buttons. Also check the stack to verify that the samples are placed at the correct addresses.

  Lab Question 3
  What happens if we do not return the allocated memory in the subroutine?

Lab Question 4
Do we need to store the values in the allocated array? What happens if we just store the values randomly on the stack?

You are now done with this part, show your work to a lab assistant! _____ ☑

Calling Conventions

The focus of this chapter is calling convention to pass arguments to and from functions and subroutines. The chapter is organized as follows. First, there is a brief introduction to argument passing, which is

Arguments Passing

When calling a subroutine, arguments must be passed to the function and a return value is (sometimes) returned. There are several ways one can pass arguments to a function. These "ways" are called *calling conventions*. It is an agreement of how to pass and return values. There are in practice two general ways this can be done: by putting all arguments on the stack, or by using registers together with the stack.

When writing purely in assembly, the programmer may choose any convention he or she desires, as long as they are being consistent. However, when mixing C and assembly code, certain rules must be followed. The C compiler will follow a specific calling convention and the code written in assembly **must** comply, otherwise the code is likely to crash.

In AVR processors, the calling convention used is the following. Arguments are placed in the registers `r25` to `r8`, using two registers each. If there are additional arguments, they are pushed on the stack. If we want to pass a `char` to a subroutine, we place it in register `r24` and `r25`. Since the size of a `char` is only one byte, we put it `r24` and we let `r25` be zero. Note that we place the lowest significant byte in the register with the lowest value. An example of different function calls and how the values are passed is shown in Table 4.

See Section 6 in `http://ww1.microchip.com/downloads/en/appnotes/doc42055.pdf`. Note that there is a typo for register `r22`. It should be `b2`, not `b2b`.

When returning a value from a subroutine, the value is placed in register `r24` and `r25`, the lowest byte placed in `r24`. If the return value is an 8-bit value, you do not have to clear register `r25`.

An example of C code calling a subroutine in assembly is shown in Listing 10 and in Listing 11 respectively.

```
extern char myadd(char, char);

int main()
{
    char a = 12;
    char b = 24;
    char c = myadd(a, c);
}
```

Listing 10: C program that calls a subroutine.

Table 4: Function calls with arguments places in registers.

Function call	Register values
`char a = 0x12;` `func(a)`	`r25` = 0x00 `r24` = 0x12
`char a = 0x1234;` `func(a);`	`r25` = 0x12 `r24` = 0x34

`char a = 0x23;`
`char b = 0x42;`
`func(a, b);`
	`r25` = 0x00
	`r24` = 0x23
	`r23` = 0x00
	`r22` = 0x42

```
.global myadd

myadd:
    ; input arguments are in r24 and r22
    add r24, r22     ; add arguments and store the result in r24 (return value)
    ret              ; return from subroutine
```

Listing 11: A subroutine adding numbers, being called from C.

Note the two keywords, `extern` in the C code, and `.global` in the assembly code. The `extern` keyword tells the compiler that the function is defined somewhere else. In this case, in the assembly file. The `.global` keyword lets the compiler know that the function shall be accessible outside the assembly file. Without this, the C program will not "see" the function.

Register Usage

All registers, `r0` to `r31` may be used in a subroutine, but some registers need to be restored because the caller function (C code) expects the values in those registers not to change, see Table 5. The registers to be saved shall be pushed on the stack in the subroutine prologue, and popped off the stack, in the epilogue.

For more details, see Section 5 in `http://ww1.microchip.com/downloads/en/appnotes/doc42055.pdf`. Note that there is a typo in the table. The next to last register should be `r30`, not `r0`.

Table 5: Register usage in assembly, when called from C code.

Register	**Usage**
`r0`	Save and restore if using
`r1`	Always 0, clear before return if using
`r2-r17`	Save and restore if using
`r28`	Save and restore if using
`r29`	Save and restore if using
`r18-r27`	Can freely use
`30`	Can freely use
`31`	Can freely use

Lab Exercises

Being limited to only write in assembly is daunting for most people (authors excluded), and most would like to write only the critical parts in assembly but the rest in a more user friendly language. Just as we can call Java functions from C, C from Python, and C from Matlab, we can of course call assembly from C (and vice versa).

In this assignment, we will explore the combination of C and assembly.

Tasks:

- Create a new "GCC C Executable" project in Atmel Studio.
- Write a program that blinks an LED if a button is pressed. That is, if a button is pressed, you shall call `led_on()` and `led_off()` with a delay after both functions, use 500 ms. Note that the two functions are not yet implemented, thus you will need to implement them in assembly. On the top of your C-file, add
  - `extern void led_on(char)`,
  - `extern void led_off(char)`,
  - `extern char check_button(char)`.

  This tells the compiler that the functions are defined elsewhere.

Remember to specify the data direction, to make the pins connected to the LEDs outputs.

Remember to define `F_CPU` to 16 MHz before including the <util/delay.h> file.

- Right-click the project name in the solution explorer and select `add -> New Item...` then select `Preprocessing Assembler File (.S)` to add a new assembly file.

Home Assignment 8
In AVR assembly, how do you declare a subroutine? How do you call it?

Home Assignment 9
In AVR assembly, how are arguments passed to and returned from a subroutine?

- Declare the three subroutines and implement them following all conventions.

Lab Question 5
When calling the subroutine, `led_on`, what is being pushed to the stack and why?

Lab Question 6
When the `ret` instruction is executed, what happens with the stack pointer?

Lab Question 7
In the `check_button` subroutine, comment out the instruction where you place the return value in `r24` and run the code. Does the LED blink? If so, why?

- Run the code in a debugger to verify that the code executes as expected.

When debugging, it is good to comment the `_delay_ms` calls to avoid waiting.

You are now done with this part, show your work to a lab assistant! _____ ☑

Appendix A

AVR Instruction Set

An excerpt of AVR assembly instructions is shown in Table A.1. Rd and Rr are registers, usually `r0` to `r31`, and Imm is an immediate value which may be given in decimal, hexadecimal or in binary. One may also use mathematical operations for the immediate value, such as "+", "-", "«" and so on.

Note that not all registers can be used in all instructions. For information regarding valid registers and how many clock cycles the instructions take, see the documentation at `https://www.microchip.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html`.

Table A.1: A short summary of useful instructions.

Category	Instruction	Operation
Arithmetic	`add Rd, Rr`	Rd = Rd + Rr
	`adiw Rd, Imm`	Rd = Rd + Imm
	`sub Rd, Rr`	Rd = Rd - Rr
	`subi Rd, Imm`	Rd = Rd - Imm
	`sbiw Rd, Imm`	Rd = Rd - Imm
	`inc Rd`	Rd = Rd + 1
	`dec Rd`	Rd = Rd - 1
	`and Rd, Rr`	Rd = Rd & Rr
	`andi Rd, Imm`	Rd = Rd & Imm
	`or Rd, Rr`	Rd = Rd | Rr
	`ori Rd, Imm`	Rd = Rd | Imm
	`eor Rd, Rr`	Rd = Rd $\oplus$ Rr
	`lsl Rd`	Rd = Rd << 1
	`lsr Rd`	Rd = Rd >> 1
Branch	`breq label`	Jump to `label` if Rd = Rr
	`brlo label`	Jump to `label` if Rd < Rr
	`brne label`	Jump to `label` if Rd != Rr
	`brsh label`	Jump to `label` if Rd >= Rr
	`rjmp label`	Jump to `label`
Subroutine	`call label`	Call subroutine `label`
	`ret`	Return form subroutine
Memory	`push Rr`	Push value of register Rr on stack
	`pop Rd`	Pop top of stack and store value in `Rd`
	`ld Rd, X/Y/Z`	Load value into `Rd` from address in X, Y or Z
	`ldi Rd, Imm`	Rd = Imm
	`lds Rd, K`	Load value into Rd from address K
	`st Y, Rr`	Store value of Rr to address in Y
	`std Y+k, Rr`	Store value of Rr to addres in Y + k
	`sts Imm, Rr`	Store value of Rr to address Imm
I/O	`in Rd, Imm`	Read from I/O device at address Imm, store in Rd
	`out Imm, Rr`	Write value of Rr to I/O device at address Imm
Other	`mov Rd, Rr`	Copy value from Rr to Rd
	`movw Rd, Rr`	Copy word from Rr to Rd
	`nop`	No operation, do nothing