VIA Calling Conventions

Introduction

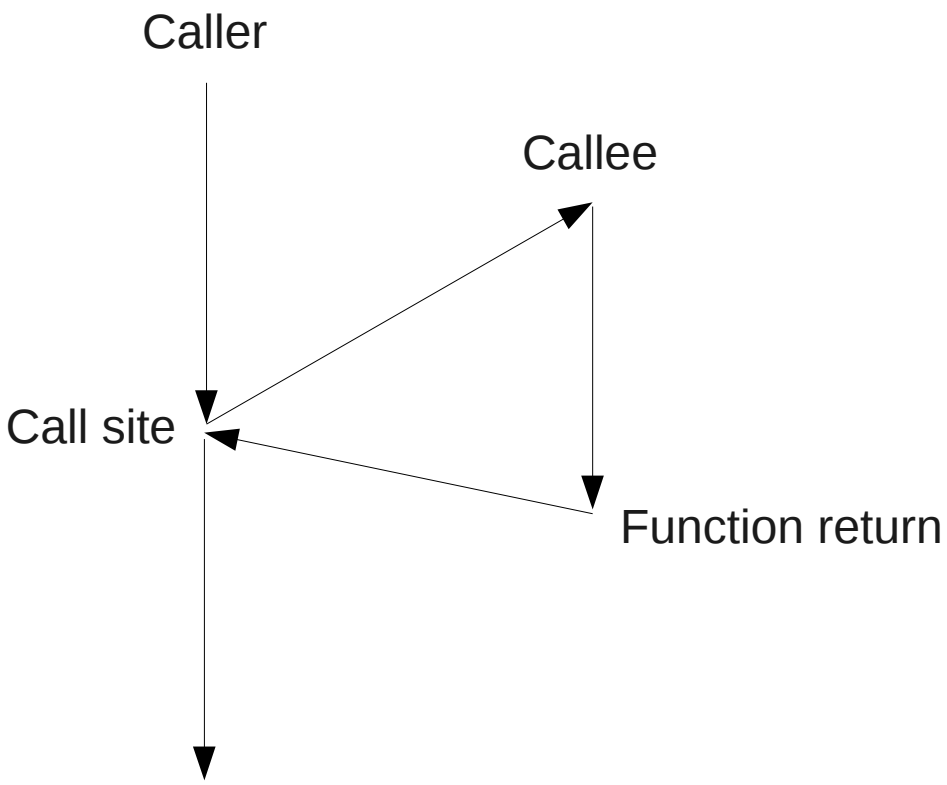
This document describes how the function concept (known as methods in Java) can be implemented in assembler. All examples are based on the ATMega2560 instruction set. The intended audience are students of the CALI1 course held at ICT Engineering, VIAUC, Horsens.

Concepts

The function concept is well known from the C programming language, and from Java as well. A function is a piece of code that solves a particular problem. The code making up the function can be called from many places and reused from many applications. A function may take zero or more values as input and return zero or one value as output.

The piece of code calling a function is termed the caller. The caller sets up the function call and executes the call. The instruction actually executing the call is termed the call site. The piece of code that implements the function itself is termed the callee. The content of all registers and machine state at the call site is termed the caller context. The content of all registers and machine state at the callee is termed the callee context. A function is said to have side-effects if the caller context is changed by the callee. Usually a function will seek to avoid having side-effects. The set of registers used by the callee is termed the working registers. A full function call will involve:

1. The caller will perform the call setup, which is preparing the input values for the call
2. The caller contains the call site actually calling the function
3. Immediately upon entry into the callee the working registers are saved
4. The callee will perform as the first thing the parameter decoding, where the input values are retreived
5. The callee implements the behaviour of the function in the function body
6. The callee restores the working registers before return
7. After proper calculation of the output value the callee returns the output values to the caller
8. After the function call returns the caller retrieves the output values at the call site



Functions in Assembler

Usually the function concept is not supported to its full extent in assembler languages. Jumping to a function and returning from the function is supported (through the CALL and RET instruction) but transferring input values and returning output values is not supported directly. The calling conventions of a particular implementation is a description of how input and output values are transferred to and from a function.

The VIA calling conventions use the stack for transferring of input and output parameters. The stack is also used by the CALL end RET instruction. All the steps required to implement a full function call in assembler is described in the following:

- Call setup. The caller uses the PUSH instruction to push the input values onto the stack.
- Call site. Using the CALL instruction the caller will jump to the entry point of the callee. The CALL instruction will push the return address onto the stack (three bytes).
- Saving working registers. As the very first thing the callee pushes all its working registers onto the stack using the PUSH instruction.
- Retrieving input values. The callee uses the X-pointer (register R26, R27) to calculate a pointer to the first input value as it is located on the stack. Using the ADIW instruction, the number of working registers, the three bytes of return address and the number of input values to the function can be added to the SP to get a pointer to the first input value. Using the LD Rx, X instruction the input values can now be loaded into the working registers.
- Implementing the function body. Code now follows in the callee to do the actual work.

- Saving output value. The result from the calculations performed in the function body is now saved on the stack at the location of the first input value. The first input value will thus be overwritten. The X pointer is set to point to the location of the first input value, and the ST X,Rx instruction is used to store the output value onto the stack.
- Restoring working registers. Just before the return the working registers are popped from the stack using the POP instruction.
- The RET instruction will return from the function to the caller.
- Back at the caller the output value is popped from the stack using the POP instruction. The number of POP instructions to execute must be the same as the number of PUSH instructions that was used to push the input values.

Functions in Assembler, an example

The following piece of code implements a delay function. The function takes two input parameters, but does not return any result. The effect of the function is to delay execution by implementing a double loop. The loop bounds on the loops are the input given to the functions.

```
1:          .INCLUDE "M2560DEF.INC"
2:          .ORG 00
3:          LDI R16, 0xFF
4:          OUT SPL, R16
5:          LDI R16, 0x21
6:          OUT SPH, R16
7:
8:          LDI R16, 0xFF
9:          OUT DDRB, R16
10:
11:       main_loop:
12:         LDI R16, 0xFF
13:         OUT PORTB, R16
14:         LDI R16, 0xFF
15:         PUSH R16
16:         LDI R16, 0x20
17:         PUSH R16
18:         CALL delay
19:         POP R16
20:         POP R16
21:         LDI R16, 0x00
22:         OUT PORTB, R16
23:         LDI R16, 0xFF
24:         PUSH R16
25:         LDI R16, 0x20
26:         PUSH R16
27:         CALL delay
```

```
28:         POP R16
29:         POP R16
30:         JMP main_loop
31:
32:       stop:
33:         JMP stop
34:
35:       delay:
36:         PUSH R16
37:         PUSH R17
38:         PUSH R26
39:         PUSH R27
40:         PUSH R18
41:
42:         IN  R26, SPL
43:         IN  R27, SPH
44:         ADIW R26, 11
45:         LD R16, -X
46:         LD R17, -X
47:
48:       outer:
49:         CPI R16, 0x00
50:         BREQ end_delay
51:         DEC R16
52:         MOV R18, R17
53:       inner:
54:         CPI R18, 0x00
55:         BREQ outer
56:         DEC R18
57:         JMP inner
58:
59:       end_delay:
60:         POP R18
61:         POP R27
62:         POP R26
63:         POP R17
64:         POP R16
65:         RET
```

This code can be run on the ATMega2560 and it will toggle output on PORTB.

Functions in assembler, exercises

Exercise 1

Identify (mark the line numbers) the following entities in the example above:

- The stack initialization
- The call setup(s)
- The call site(s)
- The callee
- The saving of the working registers
- The loading of the parameters from the stack
- The function body
- The restoring of the working registers
- The return from the function

What is the name of the function?

How many input parameters does it have?

How many output parameters does it have?

What does the function do?

Exercise 2

Using the VIA calling conventions, implement as a full and proper function the calculation of the N^{th} Fibonacci number using a iterative algorithm. The value of the 1^{st} and 2^{nd} Fibonacci number is 1. The value of the N^{th} Fibonacci number is the sum of the two previous Fibonacci numbers. As a start consider how many input values does the function take? How many output values? Maybe start by implementing the caller and then later implement the callee.

Exercise 3

Change the solution to Exercise 2 by using a recursive algorithm instead. From inside the function body you will now have to call the function itself, using our calling conventions. By some fantastic piece of magic this will work if you just follow the calling conventions without thinking too much about it.