VIA Calling Convention

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 is students of the CALI1 course held at ICT Engineering, VIA University College in Horsens, Denmark.

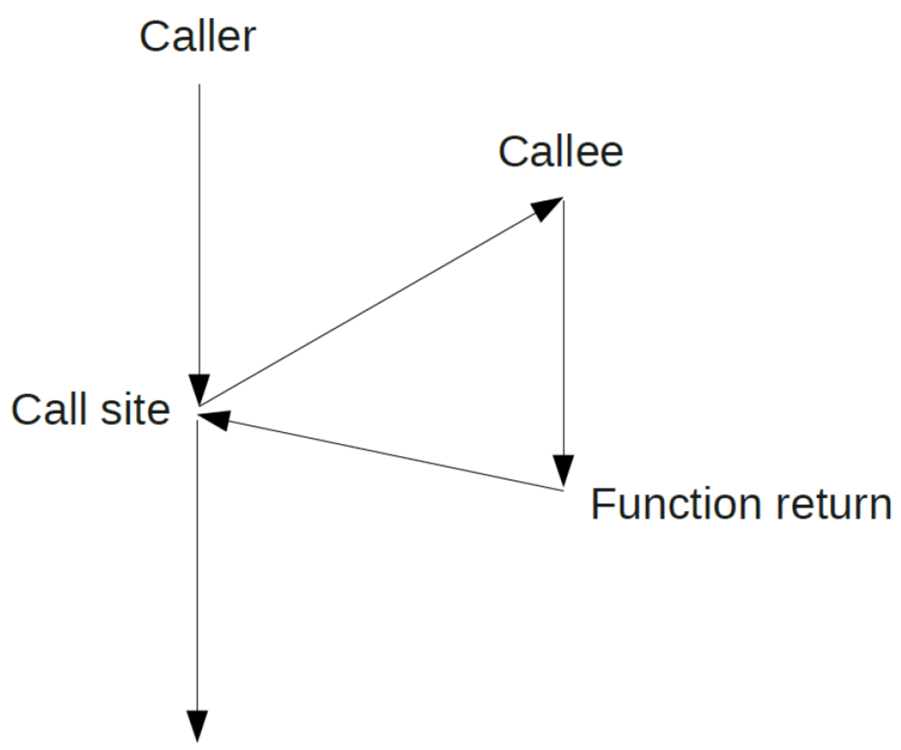
Concepts

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.

Function
Input
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 retrieved
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 convention 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:

1. **Call setup.** The caller uses the PUSH instruction to push a placeholder for the return value onto the Stack. Then it pushes the input values onto the stack.
2. **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).
3. **Saving working registers.** As the very first thing the callee pushes all its working registers onto the stack using the PUSH instruction.
4. **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.
5. **Implementing the function body.** Code now follows in the callee to do the actual work.
6. **Saving output value.** The result from the calculations performed in the function body is now saved on the stack at the location of the placeholder for the output. The X pointer is set to point to the location of placeholder for the output, and the ST X,Rx instruction is used to store the output value into the stack.

7. **Restoring working registers.** Just before the return the working registers are popped from the stack using the POP instruction.
8. **Return from the function.** The RET instruction will return from the function to the caller.
9. **Retrieving output value.** 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 and the placeholder for the return value.

Functions in Assembler, an example

The following piece of code implements a function called delay_and_invert. The function takes 1 input parameter, and return return 1 parameter. The input parameter is the number of 10ms delays (for instance will a input of 100 lead to a 1000ms delay). The output is the bitwise inverted value of portb. In the following code this function is used to toggle the test led (portb7) every second. This code can be run on the ATMega2560 and it will toggle output on PORTB.

```
.INCLUDE "M2560DEF.INC"
.ORG 00
LDI R16, 0xFF
OUT SPL, R16
LDI R16, 0x21
OUT SPH, R16

LDI R16, 0xFF
OUT DDRB, R16

main_loop:
PUSH R16
LDI R16, 100
PUSH R16
CALL delay_and_invert
POP R16
POP R16
OUT PORTB, R16
JMP main_loop

delay_and_invert:
PUSH R16
PUSH R17
PUSH R26
PUSH R27
PUSH R18

IN R26, SPL
IN R27, SPH
ADIW R26, 10
LD R18, -X


_10msdelay:
LDI R17, 160
loop2:
LDI R16, 199
loop1:
NOP
NOP
DEC R16
BRNE loop1
NOP
```

```
44:   NOP
45:   DEC r17
46:   BRNE loop2
47:
48:   DEC r18
49:   BRNE _10msdelay
50:
51:   IN r18, PORTB
52:   COM R18
53:   ADIW r26, 2
54:   ST -X,R18
55:
56:   POP R18
57:   POP R27
58:   POP R26
59:   POP R17
60:   POP R16
61:   RET
```

Exercise 1

Identify (mark the line numbers) the following 9 entities of VIA calling convention in the example above:

0. The stack initialization
1. The call setup(s)
   a. Allocation of the return value
   b. The saving of the input parameters on the stack
2. The call site(s)
3. The saving of the working registers
4. The retrieving of the input values
5. The function body
6. The saving of the return values (SKIP this as there aren't any)
7. The restoring of the working registers
8. The return from the function
9. The retrieving of the output value.

Exercise 2

Debug through the code to make sure you understand every step. When doing so, keep looking at the Stack pointer, the stack, the X-pointer and the program counter (when debugging in the simulator consider commenting out line 42, since the delay takes very long to execute in simulation mode).

Exercise 3

Using the VIA calling convention, implement a function called add7: The function takes a 8-bit number as the input and return the input plus 7.

Exercise 4

Using the VIA calling conventions, implement as a full and proper function the calculation of the N$_{th}$ Fibonacci number using an 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.