University of Victoria

Department of Electrical & Computer Engineering

CENG 255 - Introduction to Computer Architecture

Laboratory Manual

Laboratory Experiment #3

By

Farshad Khunjush, Nikitas J. Dimopoulos, and Kin F. Li

You are expected to read this manual carefully and prepare in advance of your lab session. Pay particular attention to the parts that are **bolded and underlined**. You are required to address these parts in your lab report. In particular, all items in the **Prelab** section must be prepared in a written form **before your lab**. You are required to submit your written preparation during the lab, which will be graded by the lab instructor.

Laboratory 3: Procedures and Macros

3.1 Goals

To familiarize with the conventions of calling procedures in assembly language, linking high-level and low-level codes, and to examine the difference between subroutines and macros.

3.2 Objectives

Upon the completion of this lab, you will be able to:

- Write programs that use subroutines.
- Understand the concept of ABI (Application Binary Interface).
- Distinguish the difference between Subroutines and Macros.
- Use stack frames (a.k.a. activation records) inside subroutines.
- Understand the concept of local memory allocation.

3.3 Prelab (You are required to submit your written preparation for this part, which will be graded by the lab instructor during the lab, and you have to include the graded Prelab in your final report.)

- **What is a subroutine?**
- **What is a stack?**
- **The programs in ColdFire assembly language for each part of the lab in Section 3.5 (Lab Work).**
- **A well-commented listing of each program in Section 3.5. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols used.**

3.4 Introduction

As in high-level languages, procedures (functions) are very important for writing modular, reusable, and maintainable code in assembly language.

ColdFire has several instructions to handle procedure/function calls: ***link***, ***unlink***, ***jsr*** (jump to subroutine), and ***rts*** (return from subroutine).

To implement a procedure call, the calling procedure (caller) and the called procedure (callee) must agree on certain issues:

1. How to pass the parameters.
2. How to return results (if any).
3. Where the call stack is (for local variables, saving link register, back chain, etc.).

These conventions are not part of the architecture. An *application binary interface* (**ABI**) defines everything needed for a binary object file to run on a system (CPU plus the operating system). The **ABI** includes the file format, rules for linker operation, procedure calling conventions, and register usage conventions. The **ABI** is a set of conventions to be followed by all compilers and assembly language programmers. If you follow the conventions specified by the standards in your assembly language program, it will be possible to call your procedures from procedures written in high-level languages. You will also be able to call procedures written in high-level languages from your assembly language procedures.

We describe the conventions in the ColdFire ABI in the following subsections.

3.4.1 Register Usage Convention

The following describes how the compiler uses registers.

a7	Stack pointer.
a6	Frame pointer.
d0	Returns the result of simple functions (e.g., an int)
a0	Returns the result of functions as a pointer (e.g., int*)

3.4.2 Stack Frame and Argument Passing Convention

In the ColdFire ABI, arguments are pushed onto the stack from right to left. For example, when function ***Myfoo(X,Y,Z)*** is called by a caller, the Z parameter will be pushed first, then Y, and finally X. In this case, at the beginning of the ***Myfoo*** function after performing the ***link a6,#0*** instruction the stack frame would look like Figure 1. In

ColdFire ABI, input parameters and local variables are accessed through the a6 register (the Frame Pointer). For example, inside the ***Myfoo*** function input parameter Z is accessible at 16(a6), Y at 12(a6), and X at 8(a6). The return address is located at 4(a6). If ***Myfoo*** wants to make room on the stack for local variables, it can perform a ***sub.l #<data>,a7*** to have the stack pointer, that is a7, point to the correct location for the new top of stack.

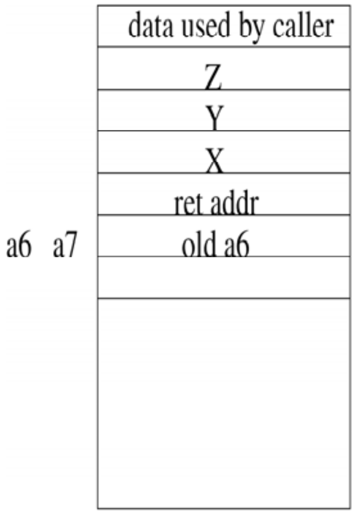

Figure 1- Stack Frame after calling Myfoo(X,Y,Z)

The following example illustrates further the details involved in calling and passing parameters to a function. We use the C programming language to show these details. As can be observed from the assembly codes in Figure 2, the a6 register is used as the Frame Pointer and the a7 register for the Stack Pointer. This figure illustrates the status of the stack in the course of program execution. In this example, we call the ***foo*** function from the ***main*** function in order to show how the assembler deals with the ***parameter passing*** and the allocation of ***local variables***.

As can be observed, we need to handle several problems. First, we have to pass input parameters to the called function through the stack. Then, inside the callee function, local variables should be allocated on the stack. We also have to save the return address onto the stack, which is accomplished automatically when the ***jsr*** function is issued; finally, we should return the result of the called function, if there is any, to the caller using registers. We describe each part in the following.

In this example the main function calls the ***foo*** function with a parameter equal to 100. Before calling the ***foo*** function (through ***jsr _foo***), the caller (i.e., the ***main*** function) pushes the input parameter (100) onto the stack. For this purpose, the caller makes room on the stack through the subtract instruction (***subq.l #4,a7***) and stores the value of 100 into the reserved space on the stack. Having pushed the input parameter, the execution of the program is transferred to the ***foo*** function through the ***jsr*** instruction.

Inside the ***foo*** function the local variable ***i*** should be allocated. For this, we make room on the stack again through the subtract instruction (***subq.l #4,a7***). This variable is accessed by ***foo*** using the a6 register, which is the frame pointer, (***-4(a6)***). Moreover, the input parameter (100) is accessed in the same way; however, the offset in this case is ***positive***. For example, the input parameter (100) is accessed through the frame pointer (***8(a6)***). It is worth mentioning that this function does not return any value to the caller; therefore, we do not need to store any result to a register. In the following example, we elaborate upon this problem.

```
void foo(int x){
        int i = x;
}
void main( ){
        foo(100);
}
_foo:       link        a6,#0        ; setup stack frame
            subq.l      #4,a7        ; make room on stack for local i
            move.l      8(a6),d0     ; x --> d0
            move.l      d0,-4(a6)    ; x --> i
            unlk        a6           ; destroy stack frame
            rts                      ; return to _main

_main:      link        a6,#0        ; a6 is used as frame pointer, save old a6
            subq.l      #4,a7        ; a7 is SP; make room on stack
```

```
moveq       #100,d0      ; 100 --> d0
move.l      d0,(a7)      ; push 100
jsr         _foo         ; foo(100)
unlk        a6           ; restore old a6
rts                      ; return form subroutine
```

Figure 2- The assembly codes for the main and foo functions

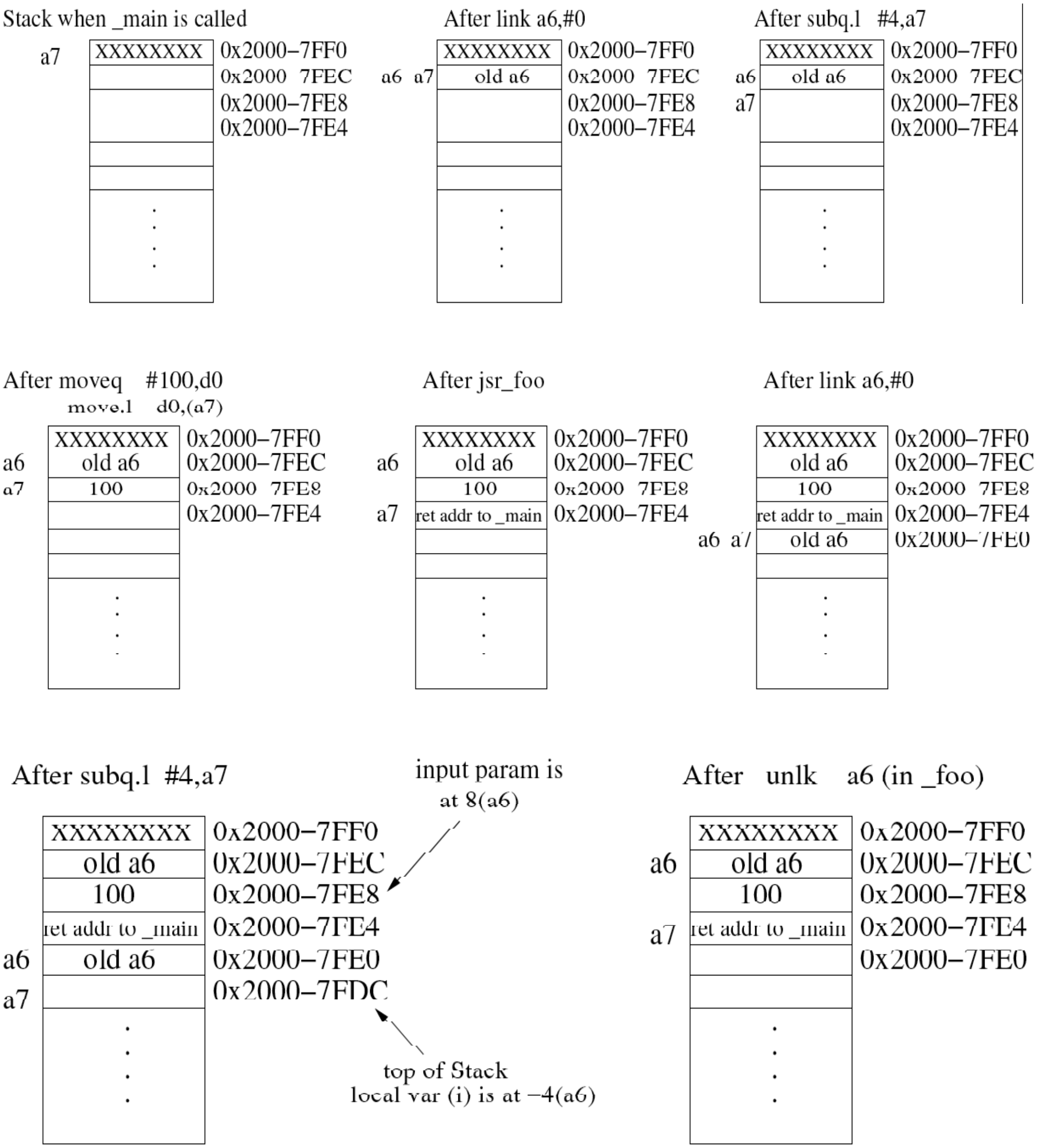

Figure 3- Stack status before and after function call

3.4.3 Returning Results Convention (Functions)

ColdFire uses registers to return results of functions. For example, the ***d0*** register is used to return simple function results (e.g., an integer). Pointers are returned in the ***a0*** register.

In this part, we provide another example that describes the steps needed to call a subroutine, which involve pushing input parameters onto the stack and continuing the execution inside the callee function (***addnumbers)***. We also demonstrate the steps needed to access the input parameters and how to return the function's result to the caller function.

The following code depicts how the input parameters are passed to the ***addnumbers*** function. As can be seen, the ***data3*** is pushed onto the stack before other parameters (i.e., ***data2*** and ***data1***). Having pushed all the input parameters onto the stack, the program continues the execution inside the ***addnumbers*** function through the ***jsr*** function.

```
;  addnumbers(data1,data2,data3);
             move.l   _data3,d0
             move.l   d0,8(a7)          ;
             move.l   _data2,d0
             move.l   d0,4(a7)
             move.l   _data1,d0
             move.l   d0,(a7)
             jsr      _addnumbers
```

This part illustrates the operations inside the ***addnumbers*** function. The callee function accesses the input parameters through the a6 register, the frame pointer. For example, the ***p1*** parameter is accessed at ***8(a6)***, ***p2*** at ***12(a6)***, and ***p3*** at ***16(a6)***. Moreover, the local variable (i.e., ***sum***) is allocated on the stack frame and is accessed at ***-8(a6)***. It should be noted that the result of this function is stored into ***d0***; therefore, the caller can access the result through the ***d0*** register.

```
int addnumbers(int p1, int p2)
{
       int sum;
       p1 = p1 + p2 * 2;
       sum = p1+p2;
       return sum;
}
_addnumbers:
             link     a6,#0
             subq.l   #8,a7      ;one temporary variable and one local variable
;
;   p1 = p1 + p2 * 2;
;
             move.l   12(a6),d0  ; p2 --> d0
             move.l   d0,-4(a6)  ; put d0 into a temp area
             move.l   8(a6),d1   ; p1 -> d1
             move.l   -4(a6),d0  ;
             lsl.l    #1,d0      ; p2 * 2 --> d0
             add.l    d0,d1      ; p1 + p2 * 2 --> d1
             move.l   d1,8(a6)   ; d1 --> p1
;
;   sum = p1+p2+p3;
;
             move.l   8(a6),d0   ; p1 --> d0
             add.l    12(a6),d0  ; add p1 and p2 --> d0
             add.l    16(a6),d0  ; add with p3 --> d0
             move.l   d0,-8(a6)  ; store d0 into sum
;
;   return sum;
;
             move.l   -8(a6),d0  ; put sum into d0 as the return value
             unlk     a6
             rts
```

3.4.5 Macro definition:

Assembler macros enable the programmer to encapsulate a sequence of assembly code in a macro definition and then in-line that code with a simple parameterized macro invocation. Therefore, each invocation of a macro causes the assembly lines in the macro definition to be added inside the code. This is the main difference between a macro and a subroutine. The subroutine lines are added just once inside a program and it does not matter how many times we invoke this subroutine. We should emphasis that there is only one copy of a subroutine. In contrast, each invocation of a macro expands the macro lines inside the program, which results in having larger source codes. All macro processing happens at ASSEMBLER time. Every time a macro is included in the body of a program, a collection of bytes representing instructions or data is included. Thus, if a macro is designed to generate 10 bytes of code, and that macro is called three times, then thirty bytes will be added to our program.

We usually use MACROs for short sequences of code, e.g., to add two numbers, to call a subroutine, or perhaps to execute a short loop. We would use a SUBROUTINE for extensive processing. This decision is related to the execution time for these two approaches. The MACRO approach has less time-overhead during execution time. Therefore, we usually use macros in the case of having short sequences of code.

Macros also allow us to do conditional processing, that is, to generate code if something is true. This is called conditional assembly. Remember, the ASSEMBLER can only check things known before the program is executed, the address of a memory location or the contents of a MACRO-variable. The conditional assembly capabilities CANNOT check the contents of a memory location; these are known only at run time and would have to be checked by code generated by the macro.

To define a macro we use the following convention. Optional parameters can be referenced in the macro body as well:

```
label:        .macro        [parameter, ….]
              macro body
              .endm
```

```
Example:
       addmacro:     .macro        reg,reg             # macro definition
                     addq.l        #data,reg
                     .endm
```

Now, if we invoke this macro as follows:

```
addmacro     3,d3         # macro invocation #1
addmacro     4,d4         # macro invocation #2
```

This will produce the following code:

```
addq.l       #3,d3        # macro expansion #1
addq.l       #4,d4        # macro expansion #2
```

3.5 Lab Work

3.5.1 In this section, you are required to **write a program in ColdFire assembly language to sort twenty signed-integer numbers residing in an array called *myarray*** (Use bubble sort for the sort algorithm with a function or procedure). You **have to** use the same structure that we introduced in the above sections.

Deliverable:

- **The program in pseudo code.**
- **A well-commented listing of your program. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols.**
- **Memory Map Assignment by the linker.**
- **Snap shots of the data section before and after execution of the program.**
- **Comparison of the execution time and the program size of this approach with the one you did in the previous lab.**

3.5.2

a) Write a recursive function to calculate the n! (n*(n-1)*(n-2)*...*2*1) for n=3.

Deliverable:

- **The program in pseudo code.**
- **A well-commented listing of your program. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols.**
- **Memory Map Assignment by the linker.**
- **The contents of the Link Register and stack status during the execution of each function call.**
- **The values of parameters and the return value in each step.**

b) Now, we would like you to investigate the overheads due to calling functions. For this,

you need to write the program, which is n!, using a simple code without using subroutines. Then, you **use the code that has been introduced in the previous lab to find out the execution time of each approach and also the size of each program in each scheme**. **Comparing the execution time and the program size of each approach, discuss the advantages and disadvantages of each scheme**.

c) Which programming style would you use if you were to calculate 15! (i.e., recursive or non-recursive)? And why?

d) What is the maximum factorial you could calculate using the recursive function you developed in (a)? How would you be able to increase this maximum?

3.5.3 In this section, **write a macro to transfer a block of memory of size N from a source location to a destination using a MACRO called *memcopy***. This MACRO has three parameters: SOURCE, DESTINATION, and SIZE. Then, **write a program that transfers N bytes from 10 different sources, whose sizes are N bytes, to 10 different destinations with the same size using the written macro, that is, *memcopy***.

Deliverable:

- **Include a well-commented listing of your program. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols.**
- **Memory Map Assignment by the linker.**
- **Snap shots of the data section before and after execution of the program.**

3.5.4 Write the same program as in 3.5.3, using procedures and compare its size and execution time with the macro approach.

Deliverable:

- **Include a well-commented listing of your program. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols.**
- **The comparison of the size and the execution time with the macro approach.**