University of Victoria

Department of Electrical & Computer Engineering

CENG 255 - Introduction to Computer Architecture

Laboratory Manual

Laboratory Experiment #2

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 **<u>bolded and underlined</u>**. 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 **<u>before your lab</u>**. You are required to submit your written preparation during the lab, which will be graded by the lab instructor.

Laboratory 2: Implementing control structures

2.1 Goal

To familiarize with the implementation of control structures using the ColdFire instruction set.

2.2 Objectives

Upon the completion of this lab, you will be able to write assembly language programs that use:

- The condition code register.
- Operations that manipulate the condition code register.
- The conditional and unconditional branching operations.

In addition, you will be able to convert control structures in high-level languages to their corresponding assembly instructions.

2.3 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 an overflow? When does it happen?**
- **What is sign-extended?**
- **What is a recursive function?**
- **What is bubble sort?**
- **The programs in ColdFire assembly language for each part of the lab in Section 2.5 (Lab Work)**
- **Include a well-commented listing of each program in Section 2.5. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols used.**

2.4 Introduction

In this lab we introduce the ColdFire branching operations and control structures so that high-level language statements can be translated into their corresponding assembly instructions. We discuss the operations for conditional and unconditional branching based on the bits in the condition-code register.

First, we introduce the condition-code register and its organization. Second, we consider the operations that affect the condition-code register. Third, we present the conditional and unconditional branching operations that use the bits in the condition-code register to control branching. Finally, we present how to translate high-level structures (e.g., ***If-then-else, while, do..while,*** and ***for***) into ColdFire assembly language.

2.4.1. The condition-code register (CCR)

The condition register in ColdFire consists of 5 bits, which holds the status of a previous operation (e.g., mathematical operations, comparisons) for a class of instructions. Many integer instructions affect the CCR that indicate some conditions of the instruction's result. Program and system control instructions also use certain combinations of these bits to control program and system flow.

Bits [3:0] represent a condition of the result generated by an operation. Bit 5, the extend bit, is an operand for multiprecision computations. Version 3 processors have an additional bit in the CCR: bit 7, the branch prediction bit.

The CCR condition-code register contains the following bits:

- The **Carry (C)** bit (CCR: bit 0). Set if a carry out of the most significant bit of the operand occurs.
- The **Overflow (V)** bit (CCR: bit 1). Set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise cleared.
- The **Zero (Z)** bit (CCR: bit 2). Set if the result equals zero; otherwise cleared.
- The **Negative (N)** bit (CCR: bit 3). Set if the most significant bit of the result is

set; otherwise cleared.

- The **Extend (E)** bit (CCR: bit 4). Set to the value of the C-bit for arithmetic operations; otherwise not affected or set to a specified result.
- 6–5 — Reserved, should be cleared.
- The **Branch prediction (P)** (Version 3 only) (CCR: bit 7).

2.4.2. Status Register (SR)

The SR stores the processor status, the interrupt priority mask, and other control bits. User software can read or write only SR[7–0], which is the “Condition Code Register (CCR).” The control bits indicate processor states: trace mode (T), supervisor or user mode (S), and master or interrupt state (M).

2.4.3. Branch Instructions

The ColdFire architecture provides a variety of branch instructions. First, we discuss unconditional branch instructions.

2.4.3.1 Unconditional branch instructions

This class of branch instructions takes place without considering conditions. We introduce different modes for unconditional branches:

- **Unconditional branch**, Relative addressing: in this case a relative displacement field encoded as a part of the instruction is added to the CPU’s program counter register. The syntax for this instruction is as follows:

  BRA target_address
- **Unconditional Jump**, Direct(Absolute) addressing: in this case the CPU’s program counter register is loaded with the address of the target location. The address resides in the instruction. The syntax for this instruction is as follows :

  JMP target_address
- **Branch to Subroutine,** Relative addressing: The **BSR** instruction is typically used to transfer the control of a program to subroutines. Besides transferring program's control to a subroutine, the PC for the next instruction is saved on the top of the stack.

BSR target_address

- **Jump to Subroutine,** Direct (Absloute) addressing: The JSR instruction is typically used to transfer the control of a program to subroutines. Besides transferring program's control to a subroutine, the PC for the next instruction is saved on the top of the stack.

  JSR target_address

- **Return from Subroutine,** The RTS instruction is used to transfer the control of a program from a subroutine to its caller.

  RTS

2.4.3.2 Conditional branch instructions

The ColdFire has a rich set of conditional branches. These instructions have direct addressing mode and allow you to test any of the conditional bits in the condition-code register (CCR). Table 1 shows the syntax for these instructions.

Table 1 - Syntax for conditional Branches

Instruction	Description	Branch condition
BCC target	Branch on carry clear	not C
BCS target	Branch on carry clear	C
BEQ target	Branch on equal	Z
BGE target	Branch on greater than or equal	N.V + not N . not V
BGT target	Branch on greater than	N.V.(not Z) + (not N).(not V).(not Z)
BHI target	Branch on higher than	not (CZ)
BLE target	Branch on less than or equal	Z + N.(not V)+(not N).V
BLS target	Branch on lower than or same	C + Z
BLT target	Branch on less than	N.(not V) + (not N).V
BMI target	Branch on minus (i.e., negative)	N
BNE target	Branch on not equal	not Z
BPL target	Branch on plus (i.e., positive)	not N
BVC target	Branch on overflow clear	not V
BVS target	Branch on overflow set	V

It should be noted that the BGE, BGT, BLE, and BLT instructions are signed comparison. However, the BCC, BHS, BHI, BLS, and BCS instructions are unsigned comparison.

Table 2 depicts different program control instructions.

Table 2 - Program Control Instructions

Instruction	Operand Syntax	Operand Size	Operation
		Conditional	
Bcc	<label>	B, W, L	If Condition True, Then PC + $d_n \rightarrow$ PC
FBcc	<label>	W, L	If Condition True, Then PC + $d_n \rightarrow$ PC
Scc	Dx	B	If Condition True, Then 1s $\rightarrow$ Destination; Else 0s $\rightarrow$ Destination
		Unconditional	
BRA	<label>	B, W, L	PC + $d_n \rightarrow$ PC
BSR	<label>	B, W, L	SP – 4 $\rightarrow$ SP; nextPC $\rightarrow$ (SP); PC + $d_n \rightarrow$ PC
FNOP	none	none	PC + 2 $\rightarrow$ PC (FPU pipeline synchronized)
JMP	<ea>y	none	Source Address $\rightarrow$ PC
JSR	<ea>y	none	SP – 4 $\rightarrow$ SP; nextPC $\rightarrow$ (SP); Source $\rightarrow$ PC
NOP	none	none	PC + 2 $\rightarrow$ PC (Integer Pipeline Synchronized)
TPF	none #<data> #<data>	none W L	IPC + 2 $\rightarrow$ PC PC + 4 $\rightarrow$ PC PC + 6 $\rightarrow$ PC
		Returns	
RTS	none	none	(SP) $\rightarrow$ PC; SP + 4 $\rightarrow$ SP
		Test Operand	
TAS	<ea>x	B	Destination Tested $\rightarrow$ CCR; 1 $\rightarrow$ bit 7 of Destination

CC—Carry clear	GE—Greater than or equal
LS—Lower or same	PL—Plus
CS—Carry set	GT—Greater than
LT—Less than	T—Always true[1]
EQ—Equal	HI—Higher
MI—Minus	VC—Overflow clear
F—Never true [1]	LE—Less than or equal
NE—Not equal	VS—Overflow set

[1] Not applicable to the Bcc instructions.

2.4.4 Standard Control Structures in Coldfire Assembly Language

In this section, we describe the implementation of standard high-level control structures in ColdFire assembly language. We explain the implementation of the followings:

- ***if-then-else*** statements,
- ***switch*** and ***case*** statement,
- **while** and ***do-while*** structures,
- ***for*** loop structures.

2.4.4.1 if statement

The basic high-level control structure is the ***if*** statement. In this section, we show how we implement the ***if*** statement in ColdFire assembly language. For this purpose, we consider the following ***if*** statement example:

A C-code example for the ***if*** statement

```
if (l1 > l2)
    l1 = l2 + 5;
else
    l1 = l2 - 10;
```

The translation for this piece of code could be as following:

```
         move.l   #l1,a1    ; &l1 -> a1
         move.l   #l2,a2    ; &l2 -> a2
         move.l   (a1),d1   ; l1 -> d1
         move.l   (a2),d2   ; l2 -> d2
if1:     cmp.l    d2,d1     ; l1 – l2 -> CCR
         ble.s    else1     ; (if  !(l1 – l2 > 0) then do the else part )
         move.l   d2,d0
         addq.l   #5,d0
         move.l   d0,(a1)
         bra.s    fi1
else1:
         move.l   d2,d0
         addi.l   #-10,d0
         move.    d0,(a1)
```

```
fi1:
```

In the first step we have to obtain the address of the operands (i.e., l1 & l2) to transfer their contents to registers for subsequent operations. Having loaded the address of the variables, their contents are transferred into d1 and d2 using **move.l** op-code. Afterwards, we compare them and perform the necessary operation. Finally, the contents of the register are stored into the corresponding variable, which is l1.

As can be observed in the last example, the contents of variables have to be loaded into the registers before manipulations, and the final result also has to be stored into the original location after the required operations. These transfers result in poor performance for applications. Therefore, optimizing compilers and assemblers try to keep the contents of the variables in registers in order to improve performance. For this reason, we use registers to represent variables' values onward.

Considering l1 is in the d1 register and l2 is in the d2 register, we have the following code for this structure.

```
if1:     cmp.l    d2,d1         ; l1 - l2 -> CCR
         ble.s    else1         ; (if  !(l1 - l2 > 0) then do the else part )
         move.l   d2,d0
         addq.l   #5,d0
         move.l   d0,d1
         bra.s    fi1
else1:
         move.l   d2,d0
         addi.l   #-10,d2
         move.l   d2,d1
fi1:
```

As can be observed from the corresponding assembly code, the tested condition in the branch instruction is the opposite of the original condition in the original ***if*** statement in the C program. It is also possible that the ***if*** structure does not have the ***else*** section. In such a case, the corresponding assembly language is shown below.

```
if   (l1 > l2)
     l1 = l2 + 5;

if1: cmp.l    d2,d1         ; l1 - l2 -> CCR
     ble.s    fi1           ; (if  !(l1 - l2 > 0) then do nothing )
     move.l   d2,d0
     addq.l   #5,d0
```

```
    move.l   d0,d1
fi1:
```

The other possibility is having a combination of conditions (e.g., using bitwise logical operations such as **&&** or **||** in C) in ***if*** statements. Some examples situations are given here.

Assumption: l1 is in d1, l2 is in d2, and l3 is in d3

```
if (l1 > l2 && l2 < l3)          if3:     cmp.l    d2,d1    # compare l1 and l2
    l1 = l2 + 5;                          ble.s    else3
else                                      cmp.l    d3,d2    # compare l2 and l3
    l1 = l2 - 10;                         bge.s    else3
                                          move.l   d2,d0
                                          addq.l   #5,d0    # then part
                                          move.l   d0,d1
                                          bra.s    fi3
                                 else3:
                                          move.l   d2,d0
                                          addi.l   #-10,d0
                                          move.l   d0,d1    # else part
                                 fi3:
```

```
if (l1 > l2 || l2 < l3)          if4:     cmp.l    d2,d1    # compare l1 and l2
        l1 = l2 + 5;                      bgt.s    then4
    else                                  cmp.l    d3,d2    # compare l2 and l3
        l1 = l2 - 10;                     bge.s    else4
                                 then4    move.l   d2,d0
                                          addq.l   #5,d0    # then part
                                          move.l   d0,d1
                                          bra.s    fi4
                                 else4:   move.l   d2,d0
                                          addi.l   #-10,d0
                                          move.l   d0,d1
                                 fi4:
```

<u>**Explore the differences between these two examples.**</u>

2.4.4.2 Switch statement

The next high-level structure of interest is the ***switch*** (or case) statement. Its purpose is to allow the value of a variable or expression to control the flow of program execution. The next example shows this structure in C language.

```
switch (x1)
    {
        case 0: l2 = l1 + 1;
                break;
        case 1: l2 = l1 + 2;
                break;
        case 2: l2 = l1 + 4;
```

```
                break;
        case 3:  l2 = l1 + 6;
                 break;
        default : l2 = l1 + 10;
    }
```

The most common way to convert a switch structure into its corresponding assembly structure is to use a chain of ***if..else..if*** statements. We use this approach to convert this structure into assembly language. The following is the equivalent assembly code for the above C code.

Assumption: l1 is d1, l2 is d2, x1 in d5

```
sw1:     cmpi.l   #0,d5
         beq.s    case0         # do case 0
         cmpi.l   #1,d5
         beq.s    case1         # do case 1
         cmpi.l   #2,d5
         beq.s    case2         # do case 2
         cmpi.l   #3,d5
         beq.s    case3         # do case 3
         bra.s    default       # do default
case0:
         move.l   d1,d0
         addq.l   #1,d0
         move.l   d0,d2
         bra.s    ws1
case1:
         move.l   d1,d0
         addq.l   #2,d0
         move.l   d0,d2
         bra.s    ws1
case2:
         move.l   d1,d0
         addq.l   #4,d0
         move.l   d0,d2
         bra.s    ws1
case3:
         move.l   d1,d0
         addq.l   #6,d0
         move.l   d0,d2
         bra.s    ws1
default:
         move.l   d1,d0
         addi.l   #10,d0
         move.l   d0,d2
ws1:
```

2.4.4.3 while and do…while loops

The ***while*** loop is an iterative statement provided in high-level languages. The ***while*** loop tests a ***boolean*** expression at the beginning of a loop body and executes the body if the expression evaluates to ***true***. The following C code and its ColdFire assembly implementation demonstrate this situation.

Assumption: a in d1 and b in d2

```
while ( a > b )          wi1:
     b += 5;                  cmp.l    d2,d1  ;  a – b -> CCR
                              ble.s    wend1  ; branch if !((a-b)>0)
                              addq.l   #5,d2
                              bra.s    wi1
                         wend1:
```

The ***do…while*** is another iterative structure in high-level languages. It differs from the **while** structure in the location of testing the condition. This structure checks the correctness of the Boolean expression at the end of the loop instead of the beginning. The next example shows this structure and its ColdFire assembly language implementation.

```
do{                      do1:
     b += 5;                  addq.l   #5,d2
}while ( a > b );             cmp.l    d2,d1    ; a – b -> CCR
                              bgt.s    do1
                         od1:
```

<u>Explain the difference between *while* and *do…while* constructs in terms of the condition that is checked at the beginning of *while* and the condition at the bottom of *do…while* constructs in assembly language.</u>

2.4.4.4 For loop

When the number of iterations is known prior to executing the loop, we use ***for loop*** in high-level languages. The next example shows this structure in high-level and the corresponding low-level implementation.

```
                                        moveq    #0,d2    # b = 0
                                        moveq    #0,d0    # i = 0
b = 0;                         for1:    cmpi.l   #100,d0  # Compare (i - 100)
for ( i = 0; i < 100; i++)              bge.s    endfor1  # i – 100 < 0 ?
        b += i;                         add.l    d0,d2    # b += i
                                        addq.l   #1,d0    # i++
                                        bra.s    for1     # go to the loop
                               endfor1:
```

2.5 Lab Work:

2.5.1 In this section, you are required to **<u>write a program in ColdFire assembly language to generate the first twenty prime numbers greater than 2 and save them into a memory location called *Primearray*.</u>** You **<u>have to</u>** use the same structures that we introduced in previous sections. For this part, you may use the following algorithm. **<u>You are required to optimize the codes in terms of its execution time and the required</u>**

space.

Note: You can use the piece of code that is introduced in the next part of the lab to find out the execution time of any part of the program.

```
  for ( i = 3, n = 0; n < 20; i+= 2){
    prime = 1; // A switch to verify that this number is a prime or not
    Limit = i / 2;
    for ( j = 2; j < Limit; j++ ){
      if ( i % j == 0 ){  // (i MOD j) == 0
         prime = 0;
         break;
      }
    }
    if ( prime ){
      n++; // The number of prime numbers so far
      printf ("%d \n", i); // print the prime number (in Assembly Write it
                           // into the above-mentioned location (Primearray))
    }
  }
```

Deliverable:

- **Your program in pseudo code.**
- **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.**
- **The execution time of your code.**

2.5.2 In this section, **write a program in ColdFire assembly language to sort twenty signed-integer numbers reside in an array called *myarray*.** (Use bubble sort for the sort algorithm without using any function or procedure). Your implementation will be ***evaluated*** in comparison to the other students' implementations based on the execution time and the required space. You need to use a piece of code, which will be provided by your TA, to measure the execution time of any part of your programs in cycles.

You **have to** use the same structure that we introduced in the sections above.

Deliverable:

- **The program in pseudo code.**

- **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.**
- **The execution time of your code.**