Laboratory Exercise 3
Assembly Language Programming, Interrupts, and the OS Interface

Goals

After this laboratory exercise, you should have some understanding of programming in assembly language and of the interface between high-level and assembly language. You should understand the basic principles of interrupts and how interrupts can be used for programming. You should also know the difference between polling and using interrupts and the relative merits of these methods.

Literature

- Patterson and Hennessy: Chapter 2.7, 2.9, 2.10, 2.13, 5.7, Appendix A.6, A.7, A.10, or
  Brorsson: Chapter 4, 5.1–5.2 Appendix D.
- MIPS Lab Environment Reference

Preparation

Read the literature and this laboratory exercise in detail and solve the home assignment. *Note that the home assignments of this laboratory exercise demand much more work than the home assignments in the previous labs.*

Home Assignment 1

Study the following (not so) simple assembly program, which calls a subroutine that finds the largest number in a vector with N elements:

```
        # Laboratory Exercise 3, Home Assignment 1
        # Written by Jan Eric Larsson, 5 November 1998

#include <iregdef.h>

        .data
        .align 2
        .globl Test

Test:   .word 1
        .word 3
        .word 5
        .word 7
        .word 9
        .word 8
        .word 6
        .word 4
        .word 2
        .word 0

TextA:  .asciiz "Lab 3, Home Assignment 1\n"
TextB:  .asciiz "The max is %d\n"
TextC:  .asciiz "Done\n"
```

```
        .text
        .align 2
        .globl FindMax
        .ent FindMax

FindMax:
        subu    sp, sp, 8    # Reserve a new 8 byte stack frame
        sw      s0, 0(sp)    # Save value of s0 on the stack
        sw      s1, 4(sp)    # Save value of s1 on the stack

        ### Add code to find maximum value element here! ###

        lw      s1, 4(sp)    # Restore old value of s1
        lw      s0, 0(sp)    # Restore old value of s0
        addu    sp, sp, 8    # Pop the stack frame
        jr      ra           # Jump back to calling routine

        .end FindMax

        .text
        .align 2
        .globl start
        .ent start

start:  subu    sp, sp, 32   # Reserve a new 32 byte stack frame
        sw      ra, 20(sp)   # Save old value of return address
        sw      fp, 16(sp)   # Save old value of frame pointer
        addu    fp, sp, 28   # Set up new frame pointer

        la      a0, TextA    # Load address to welcome text
        jal     printf       # Call printf to print welcome text

        la      a0, Test     # Load address to vector
        jal     FindMax      # Call FindMax subroutine

        la      a0, TextB    # Load address to result text
        move    a1, v0       # Move result to second register
        jal     printf       # Call printf to print result text

        la      a0, TextC    # Load address to goodbye text
        jal     printf       # Call printf to print goodbye text

        lw      fp, 16(sp)   # Restore old frame pointer
        lw      ra, 20(sp)   # Restore old return address
        addu    sp, sp, 32   # Pop stack frame
        j       _exit        # Jump to exit routine

        .end start
```

Read the literature carefully and make sure that you understand the program above in detail. Then write the missing code of the `FindMax` subroutine in assembly language. Note that arguments and results are transferred in the standard MIPS fashion, as described in the literature. In the subroutine you can use the temporary registers t0–t9 as you wish. For variables such as *n* and *Max* you can use the saved temporary registers s0–s6, but if you do, their contents must first be stored on the stack and then restored before the subroutine returns. In the code above, the subroutine `FindMax` allocates a stack frame of eight bytes and saves the old values of s0 and s1 in it. At the end, these values are restored and the stack frame deallocated again.

Home Assignment 2

Study the following assembly program, which waits for interrupts and prints out information about which interrupts it receives. Go over the code in detail and make sure that you understand everything, especially how to write and install an interrupt routine, how to enable an interrupt, and what happens when an interrupt is activated.

```
        # Laboratory Exercise 3, Home Assignment 2
        # Written by Mats Brorsson, 16 November 1998

        # This is a simple program to illustrate the idea of
        # interrupts. The interrupt routine start address is
        # 0x80000080. Only a small stub routine that immediately
        # jumps to the real interrupt routine is stored at this
        # address. The stub routine is copied to this address
        # during the program initialization.

#include <iregdef.h>
#include <idtcpu.h>
#include <excepthdr.h>

#define PIO_SETUP2 0xffffea2a

        .data

        # Format string for the interrupt routine

Format: .asciiz "Cause = 0x%x, EPC = 0x%x, Interrupt I/O = 0x%x\n"

        .text

        # Interrupt routine. Uses ra, a0, a1, a2, and a3.
        # It is also necessary to save v0, v1 and t0-t9
        # since they may be used by the printf routine.

        .globl introutine
        .ent introutine
        .set noreorder
        .set noat

introutine:
        subu    sp, sp, 22*4    # Allocate space, 18 regs, 4 args
        sw      AT, 4*4(sp)     # Save the registers on the stack
        sw      v0, 5*4(sp)
        sw      v1, 6*4(sp)
        sw      a0, 7*4(sp)
        sw      a1, 8*4(sp)
        sw      a2, 9*4(sp)
        sw      a3, 10*4(sp)
        sw      t0, 11*4(sp)
        sw      t1, 12*4(sp)
        sw      t2, 13*4(sp)
```

```
sw      t3, 14*4(sp)
sw      t4, 15*4(sp)
sw      t5, 16*4(sp)
sw      t6, 17*4(sp)
sw      t7, 18*4(sp)
sw      t8, 19*4(sp)
sw      t9, 20*4(sp)
sw      ra, 21*4(sp)

# Note that 1*4(sp), 2*4(sp), and 3*4(sp) are
# reserved for printf arguments

.set reorder

mfc0    k0, C0_CAUSE    # Retrieve the cause register
mfc0    k1, C0_EPC      # Retrieve the EPC
lui     s0, 0xbfa0      # Place interrupt I/O port address in s0

la      a0, Format      # Put format string address in a0
move    a1, k0          # Put cause in a1
move    a2, k1          # Put EPC in a2
lbu     a3, 0x0(s0)     # Read the interrupt I/O port
jal     printf          # Call printf

sb      zero,0x0(s0)   # Acknowledge interrupt, (resets latch)

.set noreorder
lw      ra, 21*4(sp)    # Restore the registers from the stack
lw      t9, 20*4(sp)
lw      t8, 18*4(sp)
lw      t7, 18*4(sp)
lw      t6, 17*4(sp)
lw      t5, 16*4(sp)
lw      t4, 15*4(sp)
lw      t3, 14*4(sp)
lw      t2, 13*4(sp)
lw      t1, 12*4(sp)
lw      t0, 11*4(sp)
lw      a3, 10*4(sp)
lw      a2, 9*4(sp)
lw      a1, 8*4(sp)
lw      a0, 7*4(sp)
lw      v1, 6*4(sp)
lw      v0, 5*4(sp)
lw      AT, 4*4(sp)
addu    sp, sp, 22*4    # Return activation record

# noreorder must be used here to force the
# rfe-instruction to the branch-delay slot

jr      k1              # Jump to EPC
rfe                     # Return from exception
                        # Restores the status register
.set reorder
```

```
        .end introutine

        # The only purpose of the stub routine below is to call
        # the real interrupt routine. It is used because it is
        # of fixed size and easy to copy to the interrupt start
        # address location.

        .ent intstub
        .set noreorder

intstub:
        j       introutine
        nop

        .set reorder
        .end intstub

        .globl start            # Start of the main program
        .ent start

start:  lh      a0, PIO_SETUP2  # Enable button port interrupts
        andi    a0, 0xbfff
        sh      a0, PIO_SETUP2
        lui     t0, 0xbfa0      # Place interrupt I/O port address in t0
        sb      zero,0x0(t0)    # Acknowledge interrupt, (resets latch)
        la      t0, intstub     # These instructions copy the stub
        la      t1, 0x80000080  # routine to address 0x80000080
        lw      t2, 0(t0)       # Read the first instruction in stub
        lw      t3, 4(t0)       # Read the second instruction
        sw      t2, 0(t1)       # Store the first instruction
        sw      t3, 4(t1)       # Store the second instruction

        mfc0    v0, C0_SR       # Retrieve the status register
        li      v1, ~SR_BEV     # Set the BEV bit of the status
        and     v0, v0, v1      # register to 0 (first exception vector)
        ori     v0, v0, 1       # Enable user defined interrupts
        ori     v0, v0,EXT_INT3 # Enable interrupt 3 (K1, K2, timer)
        mtc0    v0, C0_SR       # Update the status register

Loop:   b       Loop            # Wait for interrupt

        .end start
```

Home Assignment 3

Study the following assembly program. Whenever a button is pressed (K1 or K2), it will copy the current position of the switches on the lab board to the LEDs. At the same time, the program pretends to perform a demanding computation, in this case a long loop. Make sure you understand how the program works and why. This method of repeatedly checking for input is called polling.

```
        # Laboratory Exercise 3, Home Assignment 3
        # Written by Georg Fischer, 16 November 1998

#include <iregdef.h>
#include <idtcpu.h>
```

```
#define SWITCHES 0xbf900000
#define LEDS     0xbf900000
#define BUTTONS  0xbfa00000

        .globl start
        .ent start

start:  sub     sp, sp, 4        # Reserve new stack space
        sw      ra, 0(sp)        # Save return address

Loop:   jal     Comp             # Perform heavy computations

        la      t0, BUTTONS      # Place address of buttons in t0
        lb      a1, 0x0(t0)      # Load button port value
        andi    a1, a1, 0x30     # Mask out button indication bits
        beq     a1, zero, Loop   # Loop if no button pressed

        sb      a1, 0x0(t0)      # Clear latched value
        la      t0, SWITCHES     # Place address of switches in t0
        lb      a0, 0x0(t0)      # Load switch position
        la      t0, LEDS         # Place address of LEDs in t0
        sb      a0, 0x0(t0)      # Output switch position to LEDs
        b       Loop             # Repeat polling loop

                                 # Standard program ending, but in
                                 # this case, it will never be used
        lw      ra, 0(sp)        # Restore return address
        addi    sp, sp, 4        # Dealloacte stack space
        j       _exit            # Jump to exit routine

        .end start

        .ent Comp

Comp:   li      t0, 0xffffff     # Initialize counter value
Delay:  sub     t0, t0, 1        # Decrease counter by 1
        bne     t0, r0, Delay    # Test if ready
        jr      ra               # Return to polling loop

        .end Comp
```

Home Assignment 4

Study the following assembly program. It performs the same tasks as the program of Home Assignment 3, but is implemented using interrupts instead of polling. Add the missing code for the interrupt routine.

The subroutine `init_ext_int` enables the button port as an interrupt port. When the MIPS processor starts, a standard interrupt routine is already in place. The subroutine `install_normal_int` installs a normal subroutine, so that when an interrupt occurs, the installed routine will be called by the interrupt routine. The interrupt routine saves and restores registers. Thus, the installed subroutine can be written as an ordinary subroutine. The subroutine `enable_int` sets *interrupt mask bits* in the status register, thereby allowing the processor to handle interrupts. Finally, the subroutine `get_CAUSE` with no argument returns the contents of the cause register. If you call `get_CAUSE` and mask the result with `EXT_INT3`, you will get a non-zero result if a button was the cause of the interrupt. You have to perform the mask operation, because there are also other interrupts, and you do not want them to interfere with the function of the program. You can read more about the interrupt routines above in the *MIPS Lab Environment Reference*.

You will find the code of these subroutines in the file *interrupt.s* in the MipsIt software directory. Add this file to your project, and add the missing code below using `get_CAUSE`.

```
        # Laboratory Exercise 3, Home Assignment 4
        # Written by Georg Fischer, 16 November 1998

#include <iregdef.h>
#include <idtcpu.h>
#include <excepthdr.h>

#define SWITCHES 0xbf900000
#define LEDS     0xbf900000
#define BUTTONS  0xbfa00000

        .globl start
        .ent start

start:  sub     sp, sp, 4       # Reserve new stack space
        sw      ra, 0(sp)       # Save return address

        jal     init_ext_int    # Initialize interrupts
        la      a0, IntHand     # Install our own interrupt routine
        jal     install_normal_int
        li      a0, EXT_INT3    # Enable interrupt 3 (K1, K2, timer)
        jal     enable_int      # Enable external timer interrupts

Loop:   jal     Comp            # Perform heavy computations
        b       Loop            # Repeat loop

                                # Standard program ending
        lw      ra, 0(sp)       # Restore return address
        addi    sp, sp, 4       # Deallocate stack space
        j       _exit           # Jump to exit routine

        .end start

        .ent IntHand

IntHand:

        ### Add code for interrupt handler here! ###

        .end IntHand

        .ent Comp

Comp:   li      t0, 0xffffff    # Initialize counter value
Delay:  sub     t0, t0, 1       # Decrease counter by 1
        bne     t0, r0, Delay   # Test if ready
        jr      ra              # Return to polling loop

        .end Comp
```

Code Reordering

Study the assembly program shown below (from Laboratory Exercise 1). Due to the *pipeline architecture* of the MIPS processor, the instruction immediately after a branch or jump instruction will be executed before the branch or jump takes place. Similarly, after a load instruction, it takes one extra instruction execution before the loaded value is available in the register. This is why `nop` instructions have been added after branch, jump and load instructions. Pipelining will be explained later in the course.

```
        # Laboratory Exercise 1, Home Assignment 2
        # Written by Jan Eric Larsson, 27 October 1998

        .set noreorder
        .text
        .globl start
        .ent start

start:  lui     $9, 0xbf90  # Load upper half of port address
                            # Lower half is filled with zeros

repeat: lbu     $8, 0x0($9) # Read from the input port
        nop                 # Needed after load
        sb      $8, 0x0($9) # Write to the output port
        b       repeat      # Repeat the read and write cycle
        nop                 # Needed after branch
        li      $8, 0       # Clear the register

        .end start          # Marks the end of the program
```

Assignment 1

The assembler can reorder instructions or put in `nop` instructions automatically, to account for the effects of pipelining. Type in, build and upload the program of Laboratory Exercise 1, Home Assignment 2, and study the resulting code in memory. Use the simulator for this assignment.

Assignment 2

Next, remove all `nop` instructions from the program, build and upload it and study the result. Does this program work correctly?

Assignment 3

Finally, remove the `.set noreorder` directive, build, upload and study the result. Does this program work correctly? What has happened?

From now on, you will let the assembler take care of instruction reordering and adding of `nop` instructions. Remember this when you inspect disassembled code during debugging.

Subroutines and the Stack

In high-level languages, the concept of subroutines is important, because it allows structuring of code into smaller parts. In this laboratory exercise we will study how subroutines are supported in assembly and machine language.

Assignment 4

Study the following C program and make sure that you understand what it does, how it does it and all the C language constructions. The declaration of the vector `Test` below uses the C syntax for initialization of vector elements.

```
/*
 *  Laboratory Exercise 3, Assignment 4
 *  Written by Jan Eric Larsson, 5 November 1998
 *
 */

int Test[10] = { 1, 3, 5, 7, 9, 8, 6, 4, 2, 0 };

int FindMaxC(int Value[])
{
  int n, Max;

  Max = Value[0];
  for (n = 1; n < 10; n = n + 1) {
    if (Value[n] > Max) Max = Value[n];
  }
  return Max;
}

main ()
{
  printf("Lab 3, Assignment 4\n");
  printf("The max is %d\n", FindMaxC(Test));
  printf("Done\n");
}
```

This program contains a vector `Test` of ten integer variables initialized with ten single digit numbers in random order. Next, it contains a subroutine `FindMaxC`, which takes a vector as an input argument and loops through the vector to find the largest number. Finally, the `main` function, which is called when the program is started, prints a few messages and calls the subroutine.

Assignment 5

Create a C(minimal)/Assembler project, type in the program of Assignment 4, save, build, upload and run it. Does it run correctly?

Assignment 6

Now test the assembly program of Home Assignment 1. Create a project, build, upload and run. Use the disassembler and step facilities to debug your program, correct all bugs, and verify that it works correctly.

Assignment 7

The C compiler, GCC, can translate C programs to machine code. It is possible to investigate the result of this translation by inspecting the generated assembly code. Use the command View Assembler in the Build menu. Study the assembly code produced by GCC in Assignment 5, and make sure you understand everything. Compare the generated code with the assembly program of Home Assignment 1.

Assignment 8

Combine the C main program from Assignment 4 with the assembly `FindMax` subroutine of Home Assignment 1. Create a new project containing both a C and an assembly part, and make the necessary changes in the C and assembly source codes. Test the program and verify that it works correctly. Note that C and assembly source code files in the same project must have different names (i.e., different extensions are not enough).

In this assignment you used program parts in both C and MIPS assembly language. In what language is the program that is executed on the computer or in the simulator?

Assignment 9

Combine the assembly main program from Home Assignment 1 with the C subroutine `FindMaxC` from Assignment 4. Test the program and verify that it works correctly. Note that when using an assembly main program with a C project, you must replace the `start` label with `main`. The C routines already contain a start label, and will execute some C-specific initializations, before they call the main routine.

Interrupts

Interrupts are used to handle external events and as an interface to the operating system. In this laboratory exercise you will study how interrupts can be used and what interrupt programming looks like. You will also compare polling to using interrupts.

Assignment 10

Create a new project, type in, and build the program of Home Assignment 2. Execute the program *on the simulator* and investigate the effects of the different interrupts in detail.

- What does the cause register contain after an interrupt?
- What does the EPC contain after an interrupt?
- How does the processor know when the interrupt routine should be executed?
- Why is the code of the routine `intstub` copied to another address?
- What does it mean to enable an interrupt?
- How does the processor know which interrupts are enabled?
- Why must so many registers be saved and restored by the interrupt routine?

Polling or Interrupts

A computer can react to external events either by polling or by using interrupts. One method is simpler, while the other one is more systematic and also more efficient. You will study the similarities and differences of these methods using a simple “toy” example program.

Assume that you want a program to respond to the pressing of one of the buttons on the lab board by reading the positions of the switches and outputting a similar pattern on the LEDs. In other words, the user should be allowed to set the switches, and *at the moment* the K1 or K2 button is pressed, the pattern should be transferred from the switches to the LEDs.

At the same time, the program should also perform some time-consuming computations. In this case, these will be simulated by a long loop in which a counter is decreased to zero. The point of this “toy” program is to exemplify how a program can handle two different tasks (responding to a pressed button and performing a CPU-intensive computation) seemingly almost simultaneously.

Assignment 11

Create a new project, type in and build the program of Home Assignment 3. Execute the program on the lab computer hardware. How long does it take the program to respond to a pressed button? Why does it take the program this long to respond? What can be done to get a quicker response?

Assignment 12

Create a new project, type in and build the program of Home Assignment 4. Execute the program on the simulator. Use the single step facility to verify that the interrupt routine works as expected.

Assignment 13

Execute the program of Home Assignment 4 on the lab computer hardware. Compare it to the previous program of Assignment 11. Is there still a delay before the program reacts to the pressing of a button? Explain the difference between the properties of the two programs.

Conclusions

Before you finish the laboratory exercise, think about the questions below:

- What is demanded for C and assembly programs to be able to call each other?
- What is demanded for two different languages to be able to call each other?
- Explain the interface between high-level and assembly language.
- What are the advantages of high-level compared to assembly languages?
- What are the advantages of assembly compared to high-level languages?
- Under what circumstances is assembly programming useful?
- Can it be useful to understand machine code even if you are not using assembly language for programming?
- What is polling?
- What are interrupts?
- What are interrupt routines?
- What are the advantages of polling?
- What are the advantages of using interrupts?
- What are the differences between interrupts, exceptions and traps?