Introduction to Computer Engineering I (ECSE-221)
Assignment 4: Assembly Language Programming
Available: Nov 3, 2006
Due Date: Nov 17, 2006

Please submit this assignment by 5pm, Nov 17, 2006, on WebCT. Your assignment must consist of a Microsoft Word document entitled "Assign4-<ID>.doc" which contains the answers to Q1-Q3, and an assembly language program entitled "Q4-<ID>.s", which contains the answer to Q4. In these filenames, <ID> should be replaced by your student ID number; for example, "Assign4-609384123.doc" and "Q4-609384123.s".

Your assembly program must be fully documented: marks will be deducted if your program does not contain sufficient comments to explain what it is doing.

This assignment will be marked out of 100 points.

Assignments received up to 24 hours late will be penalized by 10%; assignments received up to 48 hours late will be penalized by 20%, and assignments received more than 48 hours late will not be marked.

Question 1 (10 points)

MIPS' native assembly code only has two branch instructions, `beq` and `bne`, and only one comparison instruction, `slt`. Using just these three instructions (along with the `ori` instruction to set *a* to 0 or 1), write (by hand) the MIPS assembly language equivalents for the following "C" code snippets, assuming that *a* is stored in register $4, and *b* is stored in register $5. Remember that in "C", if an expression is true, it evaluates to 1 and if false, it evaluates to 0.

a) `a = (a < b);` (1 point)
b) `a = (a > b);` (1 point)
c) `a = (a == b);` (2 points)
d) `a = (a != b);` (2 points)
e) `a = (a <= b);` (2 points)
f) `a = (a >= b);` (2 points)

Question 2 (15 points)

For this question, refer to the following MIPS code (see next page):

```
        .text

        .globl main

foo:    lw   $17, 0($sp)
        lw   $18, 4($sp)
        addi $sp, $sp, 8
        la   $15, arr1
        addi $15, $15, 4
        lw   $16, 0($15)
        addi $sp, $sp, -8
        sw   $17, 0($sp)
        sw   $18, 4($sp)
        jr   $31

bar:    addi $sp, $sp, -4      # Save return address
        sw   $31, 0($sp)       # by pushing on stack
        la   $12, arr1
        addi $12, $12, 8
        lw   $13, 0($12)
        addi $12, $12, 4
        lw   $14, 0($12)
        addi $sp, $sp, -4
        sw   $13, 0($sp)
        addi $sp, $sp, -4
        sw   $14, 0($sp)
        jal  foo
        lw   $14, 0($sp)
        addi $sp, $sp, 4
        lw   $13, 0($sp)
        addi $sp, $sp, 4
        lw   $31, 0($sp)       # Restore return address
        addi $sp, $sp, 4       # by popping from stack
        jr   $31

main:   la   $10, arr1
        lw   $11, 0($10)
        addi $sp, $sp, -8
        sw   $10, 0($sp)
        sw   $11, 4($sp)
        jal  bar
        lw   $10, 0($sp)
        lw   $11, 4($sp)
        addi $sp, $sp, 8
        li   $2, 10
        syscall

        .data
arr1:   .word 42, 32, 22, 12, 2
```

Note: This code is just for the purposes of this question; it doesn't do anything useful, and more to the point, is incorrect since it doesn't fully save and restore the context in the subroutines. Also, note that $sp refers to register $29, the stack pointer.

a) Assuming the code starts execution at `main`, write out which registers are pushed to and popped from the stack, in the order they are pushed and popped. For example,

push $23 to stack
push $22 to stack
pop $8 from stack
push $25 to stack
pop $25 from stack

b) Once the code has executed, what are the contents of the following registers:

$11
$13
$14
$16
$17
$18

Question 3 (25 points)

A useful exercise in understanding assembly language and its relation to machine language is to take a short assembly language program and translate it to machine language by hand.

The following program, `countbits`, counts the number of bits set to 1 in register $4 and returns the result in register $6.

```
main:       li   $10, 32            # set up loop counter
            li   $6, 0              # clear output sum

main10:     andi $12, $11, 1        # test current bit
            beq  $12, $0, main20    # skip count if not set
            addi $6, $6, 1          # otherwise increment count

main20:     srl  $11, $11, 1        # shift input right
            addi $10, $10, -1       # decrement count
            bne  $10, $0, main10    # continue until zero
            li   $2, 10             # Halt code
            syscall
```

Translate this program to machine code by hand, explaining for each line how you worked out the machine instruction.

Although it might be tempting to simply let the SPIM assembler do this, the exercise is useful way of learning the MIPS instruction formats. Refer to Appendix A of the text for descriptions of MIPS assembly language instructions and the corresponding machine codes.

Question 4 (50 points)

The program division.c is available for download as part of this assignment. It is a binary division program which works for signed integers. It contains a function, div32, which does the signed division; and a main() program which tests division for several pairs of numbers. You should download, compile, and run it to see the output.

Now, re-implement the binary division function, div32, in MIPS assembly code, assuming the following convention for passing arguments:

Register	Argument	Mechanism
$4	dividend	pass by value
$5	divisor	pass by value
$6	quotient	pass by value
$7	remainder	pass by value

Although this is not quite the convention used by a "C" compiler (quotient and remainder are pointers and would otherwise by passed by reference), we will use it here since argument passing mechanisms have not been dealt with in detail at this point in the course. In coding your function it is *absolutely* essential that the context of the calling program is fully preserved.

Then, re-implement the main() program in MIPS assembly code to test the binary division function. The SPIM environment includes a number of SYSCALL functions for printing strings and integers. Test your functions with the same arrays of test cases given. Your results should be identical.

Your assembly program must be fully documented, as follows:

First, at the start of each procedure (main and div32) you should have a list of all the registers that you use, and what each is used for. For example, you should have something like the following at the start of your div32 code (you will have many more registers, probably allocated differently; this is just to give the idea):

```
#-----------------------------------------------------------------------
#       Procedure Name:     int div32(long dividend, long divisor,
#                                     long quotient, long remainder)
#
#       Description:       <put description here>
#
#       Register Allocation:       $4:    dividend (changed)
#                                  $5:    divisor (changed)
#                                  ...
#                                  $10:   scratch
#                                  ...
#                                  $17:   dividend_sign
#                                  $18:   divisor_sign
#-----------------------------------------------------------------------
```

Second, you should have a comment containing the "C" code that you are translating just before the assembly code implementing that "C" code. You should also have a comment at the end of every line of the assembly code. Here's an example:

```
#-------------------------------------------------------------------------
#       if (dividend < 0) {
#         dividend_sign = 1;
#         dividend = -dividend;
#       }
#-------------------------------------------------------------------------

        slt    $10, $4, $0         # Check if dividend less than 0
        beq    $10, $0, div20      # If not, skip to next case
        li     $17, 1              # dividend_sign = 1
        negu   $4, $4              # negate dividend

#-------------------------------------------------------------------------
#       if (divisor < 0) {
#         divisor_sign = 1;
#         divisor = -divisor;
#       }
#-------------------------------------------------------------------------

div20:  slt    $10, $5, $0         # Check if divisor less than 0
        beq    $10, $0, div30      # If not, skip ahead
        li     $18, 1              # divisor_sign = 1
        negu   $5, $5              # negate dividend
```