EE 4720 **Homework 1** **Due: 7 February 2001**

Problem 1: Write a DLX program to reverse a C-style string, as described below. The address of the start of the string is in `r1`. The string consists of a sequence of characters and is terminated by a zero (`NULL`). The string length is not stored anywhere, it can only be determined by looking for the `NULL`. Put the reversed string in memory starting at the address in `r2`. Be sure to terminate the reversed string.

```
! r1 holds address of first character of original string.
! r2 holds address of first character of reversed string.
! Strings end with a zero (NULL) character.
```

Problem 2: The DLX program below copies a block of memory starting at address `r1` to the address `r3`, the block is of length `r2` bytes. The problem is it won't always work. Explain why not and fix the problem without unnecessarily increasing the number of loop iterations. (The program will be slower, except for special cases.) Be sure to modify the program, not a specification of what the program is supposed to do.

```
! r1 Start address of data to copy.
! r2 Number of bytes to copy.
! r3 Start address of place to copy data to.

LOOP:
 slti r4, r2, #4
 bnez r4, LOOP2
 lw   r5, 0(r1)
 sw   0(r3), r5
 addi r1, r1, #4
 addi r3, r3, #4
 subi r2, r2, #4
 j    LOOP

LOOP2:
 beqz r2, EXIT
 lb   r5, 0(r1)
 sb   0(r3), r5
 addi r1, r1, #1
 addi r3, r3, #1
 subi r2, r2, #1
 j    LOOP2

EXIT:
```

Problem 3: Implement the following procedure in DLX assembly language. The procedure is given two ways, both do the same thing, look at either one. The return address is stored in `r31`. The C `short int` data type here is two bytes (as it is on many real systems). The registers used for the procedure arguments are specified by the C variable names.

```
 void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
{
 while( size_r4-- ) *d_r3++ = *s_r1++ + *f_r2++;
}

void sum_arrays(short int *s_r1, float *f_r2, double *d_r3, int size_r4)
{
 int i;
 for(i=0; i<size_r4; i++) d_r3[i] = s_r1[i] + f_r2[i];
}
```

Problem 4: The code below contains two sets of add instructions, one in DLX assembler, the other in Compaq (née DEC) Alpha assembler. The first instruction in each group adds two integer registers, the second instruction in each group adds an integer to an immediate, the last adds two floating point registers. Information on the Alpha architecture can be found in the Alpha Architecture Handbook, `http://www.ee.lsu.edu/ee4720/alphav4.pdf`. It's 371 pages, don't print the whole thing.

```
! DLX Assembly Code
 add  r1, r2, r3  ! r1 = r2 + r3
 addi r4, r5, #6
 addf f0, f1, f2

! Alpha Assembly Code  (Destination is last operand.)
 addq r2, r3, r1  ! r1 = r2 + r3
 addq r5, #6, r4
 addt f1, f2, f0
```

Though the DLX and Alpha instructions are similar they are not identical.

- How do the data types and immediates differ between the corresponding DLX and Alpha instructions?
- Show the coding for the DLX and Alpha instructions above. Show the contents of as many fields as possible. For DLX, the `addi` opcode is 1. The `add` func field is 0 and the `addf` func field is $1d_{16}$. For the Alpha fields, see the Alpha Architecture Manual and use the following information: The *Trapping mode* should be imprecise and the *Rounding mode* should be Normal. (Trapping [raising an exception] will be covered later in the semester.)
- How do the approaches used to specify the immediate version of an integer instruction differ?
- How is the approach used to code floating-point instructions different in Alpha than DLX?