

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.

```
add r3, r1, r0    ! Copy of r1
SIZE_LOOP:
  lb r4, 0(r3)
  addi r3, r3, #1
  bnez r4, SIZE_LOOP

  subi r3, r3, #1    ! Change r3 to address of null.

REV_LOOP:
  sub r6, r3, r1
  beqz r6, EXIT
  subi r3, r3, #1
  lb r4, 0(r3)
  sb 0(r2), r4
  addi r2, r2, #1
  j REV_LOOP
EXIT:
  sb 0(r2), 0
```

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:
```

It won't work if either the source or target addresses is unaligned and at least one word copy is attempted.

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

```
andi r4, r1, #3
andi r5, r3, #3
subi r6, r4, r5
bnez r6, LOOP2    ! Word alignment of source and destination are different.
slti r7, r2, r4
bnez r7, LOOP2    ! Data ends before next aligned address.
j   LOOPBENTER
```

```
LOOPPB:                ! Copy until both addresses are word-aligned..
    lb  r5, 0(r1)
    sb  0(r3), r5
    addi r1, r1, #1
    addi r3, r3, #1
    subi r2, r2, #1
```

LOOPBENTER:

```
andi r4, r3, #3 ! r4 is zero if r3 is word-aligned.  
bnez r4, LOOPB
```

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];
}
```

```

j TEST
LOOP:
    lh  r6, 0(r1)
    addi r1, r1, #2
    movi2fp f6, r6
    cvti2d f6, f6
    lf  f8, 0(r2)
    addi r2, r2, #4
    cvtf2d f8, f8
    addd f10, f6, f8
    sd 0(r3), f10
    addi r3, r3, #8
    subi r4, r4, #1
TEST:
    bnez r4, LOOP

```

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?

The DLX integers are 32 bits, the Alpha integers are 64 bits, a size DLX does not have. The DLX floating-point data type used is IEEE 754 single (32 bits), the Alpha floating-point values are IEEE 754 double (64 bits), a type DLX does have. The DLX immediate size is 16 bits, the Alpha immediate size is 8 bits.

- 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.)

DLX:

`add r1, r2, r3`

opcode	rs1→r2	rs2→r3	rd→r1	func→add
0	2	3	1	0
0	5 6	10 11	15 16	20 21
				31

`addi r4, r5, #6`

opcode	rs1→r5	rd→r4	simm16→6
1	5	4	6
0	5 6	10 11	15 16
			31

`addf f0, f1, f2`

opcode	rs1→f1	rs2→f2	rd→f0	func→addf
0	1	2	0	0x1d
0	5 6	10 11	15 16	20 21
				31

Alpha:

`addq r2, r3, r1`

opcode	Ra→r2	Rb→r3	SBZ	l	func	Rc→r1
0x10	2	3	0	0	0x20	1
31	26 25	21 20	16 15	13 12 12 11	5 4	0

`addq r5, #6, r4`

opcode	Ra→r5	LIT→6	l	func	Rc→r4
0x10	5	6	1	0x20	4
31	26 25	21 20	13 12 12 11	5 4	0

`addt f1, f2, f0`

opcode	Fa→f1	Fb→f2	func	Fc→f0
0x16	1	2	0xA0	0
31	26 25	21 20	16 15	5 4
				0

- How do the approaches used to specify the immediate version of an integer instruction differ? In DLX immediate variants of integer instructions use a different instruction format. In Alpha the same format is used, the immediate variant is indicated by setting an immediate bit, 12.
- How is the approach used to code floating-point instructions different in Alpha than DLX?

In DLX floating-point and three-register integer instructions share the same format. In Alpha they use a different format.