EE 4720 **Homework 2** Solution **Due: 22 September 2000**

Problem 1: Compare the coding of the DLX instructions:

```
add  r1, r2, r3
addi r4, r5, #6
```

to the corresponding Sun SPARC V8 instructions:

```
add %g3, %g2, %g1   ! g1 = g2 + g3
add %g5, 6, %g4     ! g4 = g5 + 6
```

The definition of the SPARC V8 architecture is available via `http://www.ee.lsu.edu/ee4720/sam.pdf` or `http://www.sparc.com/standards/V8.pdf`. *Hint: The information needed to solve the problem is in Appendix B.*

How are the approaches used to code immediate variants of the add instructions different in the two ISAs?

In DLX the immediate variant of the add uses a different format than the two-source-register version (Type I vs. Type R). In SPARC V8 the immediate and two register adds use the same instruction type and used the same opcode, they are distinguished by a single-bit `i` field in the instruction word.

Problem 2: DLX does not have indexed addressing nor does it have autoincrement addressing. Suppose one wanted to include those addressing modes in an extended version of DLX, call it DLX-BAM (better addressing modes). The addressing modes would be used in load and store instructions. Show how they would best be coded, where the fewer changes to the coding structure the better. (For example, adding a fourth instruction type [say Type-A], would be a big change and so would be bad.) Sample mnemonics for these instructions appear below:

```
! Indexed addressing.
lw r1, (r2+r3)  ! r1 = MEM[ r2 + r3 ];
sw (r2+r3), r4  !  MEM[ r2 + r3 ] + r4;

! Autoincrement addressing.
lb r1, 3(+r2)  !  r1 = MEM[ r2 + 3 ];   r2 = r2 + 1;
lw r4, 8(+r5)  !  r4 = MEM[ r5 + 8 ];   r5 = r5 + 4;
sw 4(+r7), r8  !  MEM[ r7 + 4 ] = r8;   r7 = r7 + 4;
```

The indexed load has two source operands and a destination, so it is natural to code it as a type R instruction. The indexed store has three source operands, but like the displacement store, one of the operands can be placed in the `rd` position, and so the indexed store can also be coded as a type R instruction.

A poor solution would be to code using a modified type I format, call that type Im. The new register number would be placed in the `immed` field. This solution is poor because it adds a new type (when type R is perfectly suitable.

Problem 3: Write a C program that does the same thing as the DLX program below.

```
 ! r2: Start of table of indices, used to retrieve elements
 !     from the character table.
 ! r4: Start of table of characters.
 ! r6: Location to copy characters to.
 ! r8: Address of end of index table.

LOOP:
 lw r1, 0(r2)
 add r3, r1, r4
 lb  r5, 0(r3)
 sb  0(r6), r5
 addi r2, r2, #4
 addi r6, r6, #1
 slt r7, r2, r8
 bneq r7, LOOP
```

Solution template available via: `http://www.ee.lsu.edu/ee4720/2000f/hw02.c`

There are two solutions below, a compact one, and one that's easier to understand.

```
void
untangle(int *r2, char *r4, char *r6, int *r8)
{
  do { *r6++ = r4[*r2++]; } while ( r2 < r8 );
}

void
untangle_easy(int *r2, char *r4, char *r6, int *r8)
{
  do {

    int table_index = *r2;
    char c = r4[table_index];
    *r6 = c;

    /* Because r2 is declared int* the line below adds 4 (sizeof(int) = 4
       on Solaris 2.6) to r2. */
    r2 = r2 + 1;
    /* Because r6 is declared char* the line below adds 1 (sizeof(char) = 1
       probably by definition) to r6. */
    r6 = r6 + 1;

  } while ( r2 < r8 );

}
```

Problem 4: Re-code the DLX program above using DLX-BAM, taking advantage of the new instructions.

```
LOOP:
 lw r1, 0(+r2)
 lb r5, (r1+r4)
 sb 0(+r6), r5
 slt r7, r2, r8
 bneq r7, LOOP
```