1 Expressions in C

For a list of the operators in C, see the attached page. Expressions are built using combinations of the operators. For example:

```
x+y*z <= w << 4
```

2 Decoding Instructions

Time to decode the instructions. To do this, we need to shift and mask.

```
void decodeRType(unsigned int i)
{
    unsigned int op = i >> 26;
    unsigned int rs = i >> 21 & 0x1f;
    unsigned int rt = i >> 16 & 0x1f;
    unsigned int rd = i >> 11 & 0x1f;
    unsigned int sh = i >>  6 & 0x1f;
    unsigned int fu = i & 0x3f;
}
```

I-type and J-type can be handled in the same manner.

3 Functions

To generalise, a function declaration looks like the following

```
<return_type> <function_name>(<parameter_lists>)
{
    <body code>
}
```

return type: any legal type in C or structure.
function name: an identifier which gives the name of the function.
parameter_list: a list of expressions.

A function looks like this in general:

```
<function_name>( <expr1>, <expr2> ,.....);
```

Example:

```
unsigned int foo(unsigned int n)
{
    return n < 2 ? 1 : n * foo(n-1);
}
```

4 I/O

Very, very basic I/O requirements for this assignment. All of the "I" is handled for you in the sample code. All that is left is the "O."

In order to use the standard I/O functions, you need to use a preprocessor directive to include the "stdio.h" header file.

```
#include <stdio.h>
```

The two functions that you are likely to use are `printf` and `puts`. The behavior of `puts` is very simple, you pass it a string and prints this string to the stdout and adds a terminating newline. (`System.io.println()` in Java.)

Example:

```
puts("Hello World!");
```

The other function, `printf`, is best demonstrated by example:

```
printf("This should print out a 5: %d\n", 5);
```

the '\n' at the end of the string is a newline. The '%d' is a format specifier that gets replaced with the argument, in this case, 5. The rest of the string is printed out verbatim.

The function `printf` takes a variable number of arguments. Each argument should correspond to a format specifier. Check the man pages for various format specifiers. The few that you're likely to use are
%d: prints integers,
%u: prints unsigned integers
%f: prints double precision floating point values.

Example:

```
int x = 5;
double y = 6.5;
printf("int: %d\nfloat: %f\n", x, y);
```

Awful, yet valid, printf statement:

```
printf("%2$#-+6.3G %1$d %1$#02hhx\n", 3, 8.24);
```

5 Structs

Structs are a way to create a new data type consisting of other existing types. An example is the best way to demonstrate this:

```
struct foo
{
    int x;
    float y;
    char z;
};

struct foo f;
```

Note the ending semicolon. Also, note that the type of f is "struct foo," not "foo." To access members of f, you use the period operator:

```
f.x = 5;
f.z = 'g';
```

Note that structs can contain other structs:

```
struct bar
{
    int x;
    struct foo f;
};

struct bar b;

b.f.y = 3.0 * b.x;
```

6 Pointers

Pointers are variables that hold the address of variables (including other pointers, but that won't be used here). To get the address of a variable, you use the & operator. For example, the following code prints the address of the integer x, along with its value:

```
int x = 5;
printf("%#x: %d", &x, x);
```

To declare a pointer to an integer, you would use:

```
int *p;
```

To assign the address of x to p, you use the & operator:

```
p = &x;
```

To assign/use a value to/at the variable pointed to by p, you use the * operator to dereference the pointer.

```
printf("%d\n", *p);
*p = 5;
```

You can also do pointers to structs as in: struct foo *fp;

If struct foo is the struct previously defined, then we can access its members by first dereferencing fp and then using the period operator as in:

```
(*fp).x = 5;
```

The parenthesis are needed since '.' has a higher precedence than '*'.

As a short cut, you can use the `->` operator:

```
fp->x = 5;
```

This does exactly the same thing as above.

7 Pass by Value

All parameter passing in C is done by "pass by value." Let's give an example:

```
struct foo
{
    int x;
    int y;
};

void bar(struct foo s, struct foo t)
{
    s.x = t.x;
}

int main()
{
    struct foo f1 = { 1, 2 };
    struct foo f2 = { -2, 3 };

    bar(f1, f2);

    printf("%d\n", f1.x);
}
```

What gets printed?

Clearly this isn't what we wanted so we need to do something else. That something else is pass by pointer (note that pass by pointer is still pass by value, it's copying the value of the pointer to the local variable). Let's rewrite the two functions:

```
void bar(struct foo *s, struct foo *t)
{
    s->x = t->x;
}

int main()
{
    struct foo f1 = { 1, 2 };
    struct foo f2 = { -2, 3 };

    bar(&f1, &f2);

    printf("%d\n", f1.x);
}
```

This does what we want.