Code Efficiency

As we have already discussed, embedded systems usually have restricted resources due to factors such as weight, cost, power and they often have to meet real-time constraints. As a consequence they need to be efficient in their operation. One of the factors that can affect these is how the code that has been written for a particular embedded systems application, actually executes on the hardware.

When developing code it is natural to produce code that is quick for the coder to write, looks neat and is as short as possible. However, this does not necessarily mean that it executes well on the hardware, for example, in terms of speed or memory usage. Therefore it is worth considering how efficient code can be written so that both a program and its associated hardware can be optimised for cost, weight, etc..

This document covers some of the basics about writing efficient code which is tailored towards a specific application. However texts such as Wolf (2001) in the recommended reading list, cover this in more depth and include more advanced techniques and tools to optimise certain parts of an embedded systems performance, such as power consumption.

1 Simplifying Expressions

There are often many ways we can write statements in code. However some make more efficient use of a system's resources. One of the most common is algebraic simplification of statements. For example, the following statement requires two multiplications, one addition and intermediate storage of the first statement which is multiplied.

```
a×b + b×b
```

However, it is easily simplified to the following statement which requires only one addition, one multiplication and intermediate storage of the addition.

```
b×(a+b)
```

2 Dead Code Elimination

In some cases a programmer may introduce special debugging code into a program. In the final program this code will never be executed and can be removed to minimise the size of the program. In other cases copy statements are introduced to copy a variable to another variable. It is often the case that one of these variables can be removed and replaced with the other, i.e. if possible re-use variables instead of creating duplicates.

3 Procedure Inlining

It is often the case that call functions are used when a piece of code is frequently used or is more suited to a standalone function. Call functions can reduce the size of a program be eliminating repeated code. However, it might not be efficient to repeatedly call a function which contains only a short piece of code. In these cases it might be better to put the code "inline" in the main program body to eliminate the repeated calls. This will speed up execution but increase the size of the program and potentially interfere with the efficient operation of the memory. Some programming languages, such as C++, have function options which automatically place the code for a called function inline during compilation.

For example, if the following function is defined:

```
[a, b] = ssadd(c,d)
{
     a = 20 x c^2
     b = a + d
}
```

The main code could be written as follows:

```
:
get   xnew
[xm, xs] = ssadd(xnew, xs)
:
get   xnew
[xm, xs] = ssadd(xnew, xs)
:
get   xnew
[xm, xs] = ssadd(xnew, xs)
:
```

Alternatively, the main code could be written with the function inline:

```
:
get   xnew
xm = 20 x xnew^2
xs = xs + xm
:
get   xnew
xm = 20 x xnew^2
xs = xs + xm
:
get   xnew
xm = 20 x xnew^2
xs = xs + xm
:
```

4 Loop Transformations

Loops are a good way to compactly code repeated instructions. However they may not be an efficient way to execute some code. For example, in a repeated loop the iteration value and exit condition need to be computed during each execution of the loop. This adds processing overheads. If a loop is repeated a fixed number of times it may be more computationally efficient to “Unroll” the loop and repeat the code in the loop directly in the program (see the example below). This eliminates the need to update an iteration variable and test an exit condition. However it can significantly increase the length of a program and interfere with the efficient operation of memory.

For example:

```
for (ii = 1; i < 5; ii++) {
     a[ii] = b[ii]*ii;
}
```

In this example using a FOR loop the variable *ii* is both checked to see if it is less than 5 and incremented in each loop.

Alternatively you could instead use the following code that removes these overheads associated with the FOR loop:

```
a[1] = b[1]*1;
a[2] = b[2]*2;
a[3] = b[3]*3;
a[4] = b[4]*4;
```

Other options to speed up loop execution are to “Fuse” multiple loops into a single loop and to “Tile” loops so that loops are nested together.