Numerix-DSP **Numerix - DSP Programming Guidelines** SigLib

Return To Home Page | SigLib DSP Library

Introduction

There are many practical issues to consider when implementing DSP applications, including the choice between fixed or floating-point devices or coding in a high level language or with assembly code.

One of the most popular techniques for developing DSP systems is to simulate the system in C on a general purpose micro-processor and then port the C code onto a DSP device. For many applications, C provides perfectly acceptable performance but to achieve this, even the most modern compilers require the assistance of the programmer. The following is a list of suggestions that can make C coded real-time routines as efficient as possible.

Local Variables

- C places local variables on the stack and hence they are accessed indirectly and therefore slowly. It is often more efficient to place variables on the heap and there are two primary techniques for doing this. The first one is to declare them as globals (outside of the scope of a function) and the second technique is to declare the variable as static, within the function.

- Most compilers allow a level of optimization that will place local variables in registers however the compiler can often be assisted by explicitly declaring frequently used variables as *'register'* types.

- Re-use local variables declared as *'register'* within a function, for multiple non-conflicting variables. On processors with a small number of registers or in complex functions, the benefit will be that fewer registers need to be pushed onto the stack but the down side is that the code can become less readable.

Function Calling

- As standard, all C function parameters are placed on the stack however many compilers allow the optimization of this task by the use of an optional register based parameter model.

- Declaring functions as *'inline'* can completely remove the function call overhead but does increase the size of the object code. Compilers often incorporate a command line switch to enable the automatic in-lining of functions that are smaller than a given size.

- Always prototype functions because most compilers can use the information to optimize the code.

- Some compilers can perform further levels of optimizations if the parameters are placed in certain orders (Usually separaating the order of pointers, floating point and fixed point variables etc.).

Interrupt Service Routines

- When implementing interrupt service routines (ISRs), all registers that are used within the ISR

must be pushed onto the stack to prevent side effects. In some cases using higher levels of optimization and hence extra registers for interrupt service routines may actually slow them down, due to the overhead of the extra stack manipulation that is required at both the start and end of the ISR. Experiment with different levels of optimization on different sections of code by splitting them into separate source files.

- Variables shared between ISRs and other functions should be declared as *'volatile'* to prevent them being removed by the optimizer.

Data Types

- When any DSP functions are implemented on fixed-point DSPs it is imperative that careful attention is paid to such issues as overflow and wrap-around due to the hardware numerical bounds.

- Never use data words longer than necessary and try to ensure they can be loaded into the CPU core in a single cycle.

Loop Optimization

Most DSP algorithms, by their nature, consist of tight looped code and there are many steps that can be taken to optimize loop execution, including :

- Move constant expressions outside the loop and pre-calculate the result. Modern compilers are usually able to do this automatically however it is often better, especially with long loops, to assist the compiler by implementing this at the source code level.

- Replace division operations with multiplication by the reciprocal and if the divisor is a constant take the reciprocal operation outside the loop. Care should be taken with this route because the numerical errors will be different for each technique.

- Unrolling loops : The loop code should be repeated several times at the source code level, within the loop construct. On some processors this can benefit the performance by allowing parallel operations from separate iterations however it is important to ensure that the code section is not larger than the on-chip cache. Some DSP compilers generate single instruction looped code that is un-interruptable, unrolling an inner loop will require more program memory but the code will often be just as efficient but it will also be interruptable.

- Reduce data dependencies : By separating the data used in one operation from the data used in another parallel operation, the compiler can often utilize the on-chip resources more efficiently.

- Always try to avoid calling functions within loops or if absolutely necessary use function pointers, especially if the function that is called is data dependent.

- Analyze the performance of the compiler with respect to *'do'*, *'do while'* and *'for'* loop efficiencies. For a given algorithm, the efficiency of each technique can be both compiler and algorithm dependent.

- Try to avoid 'test and branch' operations because they can be time consuming and can call code that is not currently resident in the cache. This can often be achieved be splitting the loop into multiple instances, each handling separate conditions.

- Many compilers will perform better if the data is read from memory at the beginning of the loop and written back at the end.

Numerical Operations

- Multiply and division of integers by numbers which are powers of 2 can be usually be performed more efficiently using bit shift operations.

- Try to avoid using trigonometric functions by using look-up tables, especially in FFT routines etc.

- When using a floating point device, try to use floating point data formats, this will reduce the burden on fixed point processing units, which will probably also be required to perform the loop counting operations. Some general purpose devices can perform floating point data operations quicker than fixed point. Floating point data also has the advantage that scaling issues are less demanding and can usually be accounted for with less overhead.

- Try to avoid underflows or overflows of the numerical system, unless the algorithm demands it.

Memory Accessing

- Most compilers allow the use of different memory models however it is always better to use the smallest model necessary because large models often entail an overhead for manipulating memory segment or page pointers.

- Most modern processors include zero overhead pointer manipulation and this can mean that using pointers to access arrays in a linear fashion is often faster than using array indexing. It should be noted that this is not always true and will very from processor to processor.

- Most DSPs incorporate functionality for zero overhead looping and bit reversed addressing and in order to use these techniques it is often necessary to correctly align the base element in a data vector. Incorrect array alignment is one of the most common reasons for DSP code not working correctly.

- The CPU must access memory for loading both program instructions and data and huge benefits can be gained by analyzing the data flows and locating the heaviest loaded functions or arrays in on-chip memory. It is often a good idea to experiment with different combinations of data, stack and/or program instructions within the on-chip memory.

- Always enable the caches.

- Some DSPs have separate program and data memory spaces and on others they are combined. Pipeline and internal bus conflicts can mean that paarticular arrangements for the partitioning of program and data can be more efficient than others. See pipeline conflicts section.

- Utilize the maximum width of the external bus. Many DSPs can now load multiple parallel data words and separate them within the CPU, with no processing overhead. E.G. load two 16 bit words with one 32 bit transfer. This may require some loop unrolling.

- For efficiently performing multiple accesses to arrays and complex structures, data should be loaded into temporary local (preferably *'register'*) variables.

- For large data sets or large programs, it can be more efficient to store all the instructions and / or data in external memory and use the on-chip DMA controller to read the appropriate parts into internal memory when needed.

- In 'paged' memory systems it is often beneficial to ensure, where possible, that individual data sets do not span page boundaries because this can cause delays to be inserted in the memory access cycle.

CPU Utilization

- Use intrinsic functions. Intrinsic functions are C like functions that directly map to the low level instructions of the CPU. Often the use of these functions allow specific or more efficient variations of standard mathematical operations (E.G. +-/x).

Avoid Pipeline Conflicts

Most DSP CPUs are fed with instructions and data in a pipeline and before attempting to obtain the maximum performance from these devices it is important to be familiar with the pipeline. The Users Guides for the DSPs usually incorporate an important chapter on this subject.

- Correct partitioning of program and data across the various memory segments is critical.

- Avoid internal or external memory access conflicts.

- It is often only possible to access both internal and external memory within a single instruction cycle if the external memory access is initiated first.

Referrences

- Techniques For Developing Software For The Next Generation DSP Devices, John Edwards and Edward Young, DSP World/ICSPAT, 1997.

These suggestions are included for information and are by no means definative descriptions. If you would like to add anything to these pages then please feel free to email me and I will add your information, with my thanks.

Download this file in Adobe Acrobat format 

Return To Home Page | SigLib DSP Library

If you have any comments or questions please email Numerix : numerix@numerix-dsp.com