C6x Code Development Flow

1 Obtain/Write, Compile, Validate Native C Code on Workstation

The purpose of this step is simply to obtain standard portable C code according to specifications. This should be compiled and validated with any available test vectors on the workstation (PC, Sun, HP, etc).

2 Compile Native C Code with the C6x Compiler

- Compile and verify test vectors with C6x compiler with -o switch to invoke software pipelining and -mw switch to invoke software pipeline loop feedback.
- Profile to determine the important, high MIP areas of the code.
- Try -pm, -o3, and -mt options to improve loops.
- Declare all read only variables as const.
- Make sure all 32 bit values are defined as int (not long) and 16 bit values are defined as short.
- Define all loop counters to be int.
- Note memory bank hits (with rel 1.1 of simulator) as these might need to be avoided by using linear assembly.
- For any important area which does not satisfy performance, proceed to step 3.
- For loops where code size needs to be improved proceed to step 7.

3 Optimize C Code for C6x

- Use intrinsics to defined specific C6x operations.
- Use intrinsics and casting short arrays to int arrays to enable word access for short data.
- Use _nassert intrinsic to pass loop count information where necessary.
- Unroll loops when C6x resources are unbalanced, i.e. when there are 3 multiplies in one iteration 2 cycles are required, but 6 multiplies requires only 3 cycles.
- Profile to determine improved performance, if any.
- Note memory bank hits (with rel 1.1 of simulator) as these might need to be avoided by using linear assembly.
- For any important area which does not satisfy performance, proceed to step 4.
- For loops where code size needs to be improved proceed to step 7.

4 Write Linear Assembly

- Write C6x linear assembly without functional units using variable names.
- Add mptr directives as necessary for avoiding memory bank hits.
- Draw dependency graph to determine if there is a live-too-long, loop carry path, or split-join path problem.
    - Split-join path - insert a move or unroll loop.
    - Live-too-long problem (also specified in loop feedback) - unroll loop.
    - Loop carry path - determine if minimum loop carry path has been achieved.
- Check to see if partitioned resource bound is greater than resource bound. This implies a partitioning problem so go to step 5.
- Determine if MII has been achieved (shown in loop feedback).
- For any important area which does not satisfy performance, proceed to step 5.
- For loops where code size needs to be improved proceed to step 7.

5 Add Partitioning Information

- Add partitioning to the linear assembly. Try to split the dependency graph to balance operations on each side and also limit the number of cross paths used.
- If MII has still not been achieved try setting all functional unit values, balancing operations among individual units as evenly as possible.
- If II is still not good enough, contact tools support for more help.
- For loops that have an important outer loop which is not software pipelined, proceed to step 6.
- For loops where code size needs to be improved proceed to step 7.

6 Software Pipeline Outer Loop

- Future versions of the tools will software pipeline outer loop automatically. For now, this will have to be done by hand if necessary.
- Also, combining loop setup with prologue will be done by the tools in the future but for now this will have to be hand coded if necessary.
- For loops where code size needs to be improved proceed to step 7.

7 Reduce Code Size

- Future versions of the tools will reduce prologue and epilogue automatically. For now, this will have to be done by hand if necessary.

All of the techniques and steps described above are documented in the Programmer's Guide. It is highly recommended that you read the Programmer's Guide, especially the Optimizing C and Optimizing Assembly chapters.