‘C6x Optimization Checklist

Because most of the MIPs in many DSP applications occur in tight loops, it is important for the C6x code generation tools to be able to make maximal use of all the hardware resources in important loops. Fortunately, loops inherently have more parallelism because there are multiple iterations of the same code executing with limited dependencies between each iteration. Through a technique called software pipelining, the C6x code generation tools are able to efficiently use the multiple resources of the VelociTI architecture and obtain very high performance. Example 1 shows the code development flow recommended to achieve the highest performance on loops:

Phase	Description
1	Compile and Profile native C code • Validates original C code • Determines which loops are most important in terms of MIPS requirements
2	Add const declarations and loop count information • Reduces potential pointer aliasing problems • Allows loops with indeterminate iteration counts to execute epilogs
3	Optimize C code using intrinsics and other methods • Facilitates use of certain C6x instructions not easily represented in C • Optimizes data flow bandwidth
4a	Write linear assembly • Allows control in determining exact C6x instructions to be used • Provides flexibility of hand coded assembly without worry of pipelining, parallelism, or register allocation • Can pass memory bank information to the tools
4b	Add partitioning information to the linear assembly • Can improve partitioning of loops when necessary • Can avoid bottlenecks of certain hardware resources

Example 1. Code Development Phases

If at any phase in the flow, the desired performance is achieved, there is no need to move to the next phase. Each of the phases in the development involve passing more information to the ‘C62xx tools. Even at the final phase, development time is greatly reduced from that of hand coding, and the performance is approaching the best that could be achieved by hand.

Internal benchmarking efforts at Texas Instruments have shown that most loops achieve maximal throughput after phases 1 and 2. For those that do not, the C compiler offers a rich set of optimizations that can allow more optimization and fine tuning all from the high level C language. Finally, for the few loops that need further optimizations, there is the Linear Assembly Optimizer, which was developed to give the programmer more flexibility than C can offer, still work with in the framework of C, and be much like programming in higher level C. For more information on the Linear Assembly Optimizer, refer to the Optimizing Assembly chapters in the Programmer’s Guide and the Optimizing C Compiler User’s Guide. For example linear assembly files look in the demo directory included with the C6x tools.

In order to aide the development process a Feedback Option (-mw) is included in the code generation tools. Example 2 shows output from the Compiler and/or Assembly Optimizer of a particular loop. Refer to section 3.2.2 in the Optimizing C Compiler Book for more information.

```
;*----------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Loop Carried Dependency Bound : 3
;*      Unpartitioned Resource Bound  : 3
;*      Partitioned Resource Bound(*) : 4
;*      Resource Partition:
;*                              A-side  B-side
;*      .L units                   0       0
;*      .S units                   2       2
;*      .D units                   2       2
;*      .M units                   2       2
;*      .X cross paths             4*      3
;*      .T address paths           2       2
;*      Long read  paths           1       0
;*      Long write paths           0       0
;*      Logical  ops (.LS)         4       1
;*      Addition ops (.LSD)        2       1
;*      Bound(.L .S .LS)           3       2
;*      Bound(.L .S .D .LS .LSD)4*         2
;*
;* Searching for software pipeline schedule
;*  at  ii = 4  Failed register allocation
;*      ii = 5  Schedule found with 4
;*              iterations in parallel
;*      Done
;*----------------------------------------*
```

Example 2. Example Feedback

This Feedback is key in determining which optimizations might be useful for further improved performance. The following checklist is provided as a quick reference to techniques that can be used to optimize loops with references to more detailed documentation:

C6x Optimization Checklist

Feedback	Solution
Loop Carried Dependency Bound is much larger than Unpartitioned Resource Bound	C Code ✓ Use -pm program level optimization to reduce memory pointer aliasing *(Ref 3.2.2.2 in PG)* ✓ Add const declarations to all pointers passed to a function that are read only *(Ref 3.2.2.1 in PG)* ✓ Use -mt option to assume no memory pointer aliasing *(Ref 3.2.2 in PG)* ✓ Declare loop counters and array indexes as type int *(Ref 3.1.1 in PG)* Linear Assembly ✓ Make sure instructions accessing memory at the beginning of the loop do not use the same pointer variables as instructions accessing memory at the end of the loop *(Ref 5.6.3 in PG)*
Partitioned Resource Bound is higher than Unpartitioned Resource Bound	✓ Write code in Linear Assembly with partitioning/functional unit information *(Ref 5.3.4 in PG)*
Too many instructions, or inefficient instructions were generated by the compiler	✓ Use intrinsics in C code to pick force more efficient C6x instructions *(Ref 3.3.1 in PG)* ✓ Write code in Linear Assembly to pick exact C6x instruction to be executed *(Ref 4.3 in CG)*
Failed to software pipeline due to Register live too long	✓ Write Linear Assembly and insert MV instructions to split register lifetimes that are live too long *(Ref 5.9.4.4 in PG)*
Failed to software pipeline due to register allocation	✓ Try splitting the loop into two separate loops ✓ If multiple conditionals are used in the loop register allocation of the condition registers could be the reason for the failure. Try writing linear assembly and partition all instructions writing to a condition register evenly between the A and B sides of the machine. If there are an uneven number put more on the B side as there are 3 condition registers on the B side and 2 on the A side.
T address paths are Resource Bound	C Code ✓ Use word access for short arrays - declare int * adn use mpy intrinsics to multiply upper and lower halves of registers *(Ref 3.3.2 in PG)* ✓ Try to employ redundant load elimination technique if possible *(Ref 5.10 in PG)* Linear Assembly ✓ Use LDW/STW instructions for accesses to memory *(Ref 5.3 in PG)*
There are memory bank conflicts (specified in the memory analysis window of simulator)	✓ Write Linear Assembly and use the .mptr directive *(Ref 5.11 in PG)*
Large outer loop overhead in nested loop	✓ Unroll inner loop *(Ref 3.3.3.4 & 5.8 in PG)* ✓ Make one loop with outer loop instructions conditional on an inner loop counter *(Ref 5.13 in PG)*
Uneven resources (i.e. 3 multiplies per loop iteration)	✓ Unroll loop to make even number of resources *(Ref 5.8 in PG)*
Two loops are generated, one not software pipelined.	✓ Use _nassert statement to specify loop count info *(Ref 3.3.3.3 in PG)*
Loop will not software pipeline for other reasons	✓ Make sure there are no function calls, branches to other code, or conditional break statements in loop *(Ref 3.3.3.5 in PG)* ✓ Try making the loop counter downcounting and declared an int in C *(Ref 3.1.1 & 3.3.3.1 in PG)* ✓ Remove any modifications to the loop counter inside the loop *(Ref 3.3.3.5 in PG)*

PG - Programmer's Guide CG - C Compiler User's Guide