User's Manual, V1.4, December 2003

TriCore™ Compiler Writer's Guide

32-bit Unified Processor

Microcontrollers

Never stop thinking.

Edition 2003-12

**Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany**

User's Manual, V1.4, December 2003

TriCore™ Compiler Writer's Guide

32-Bit Unified Processor

Microcontrollers

Infineon technologies

Never stop thinking.

TriCore™ Compiler Writer's Guide

Revision History:	**2003-12**	V1.4
Previous Version:	-	

Page	Subjects (major changes since last revision)
all	Updated to include TriCore 2
all	V1.2 Comparisions between Rider A and Rider B removed
all	V1.3 TC2 References
all	V1.4 Sections 3.3 to 3.10 revised. New FPU sections 1.5 and 2.1.2.1 added.

Table of Contents **Page**

Table of Contents **Page**

Preface

This document is intended as a supplement to the TriCore architecture manual, for use by compiler writers and code generation tool vendors. It presumes a degree of familiarity with the architecture manual, although it can be read by experienced compiler writers as an "inside introduction" to the architecture.

The document is divided into the following chapters:

Chapter 1

A compendium of code generation strategy recommendations and target-specific optimizations that apply to the TriCore architecture. Many of the items in this chapter will be obvious to any experienced code generator developer who has studied the TriCore architecture. However they provide a useful checklist and potential source of design ideas. Reviewing them may also help to provide insight into the architecture and speed of the re-targeting effort.

Chapter 2

Deals with the specifics of particular TriCore implementations, as well as compiler strategies for supporting various configurations and the impact on code generation.

Chapter 3

Discusses "advanced" optimization methods. These are mostly related to extraction of parallelism from source code and loop restructuring to exploit the packed data operations of the architecture. These optimizations require control and data dependence analysis, and other techniques that are beyond the reach of "typical" commercial compilers.

Chapter 4

Covers DSP support. It covers C language extensions that allow users access to the DSP capabilities of the TriCore architecture.

Appendix A

Provides lists of packed arithmetic instructions. It is a comprehensive list of Tricore packed arithmetic instructions corresponding to each regular DSP instruction.

Appendix B

Provides some non-trivial coding examples that illustrate how to exploit the TriCore for achieving high code density.

1 Optimization Strategies

Most of the optimization strategies described in this chapter are equally applicable to the 1.2, 1.3 and 2.0 implementations of the TriCore Instruction Set Architecture (ISA). Where differences exist, they are noted.

1.1 Using 16-bit Instructions

To achieve high performance with high code density, the TriCore architecture supports both 16-bit and 32-bit instruction sizes. There is no mode bit, and no requirement for word alignment of the 32-bit instructions, so the two instruction sizes can be freely intermixed. To minimize code size, compilers must tune their register allocation and code generation strategies to take maximum advantage of the short instruction forms.

1.1.1 Register Allocation and Code Scheduling for Short Instructions

In the TriCore instruction set, the most common arithmetic and logical instructions have 16-bit coding forms that are dyadic: the left operand serves as both source and destination. Those instructions include *ADD, ADDS, AND, MUL, SUB* and *SUBS*. There are also 16-bit dyadic forms for *SH* and *SHA*, when the right operand (shift amount) is a signed 4-bit constant. In addition, there are a handful of 16-bit monadic instruction forms (single source/destination register). They include logical complement and arithmetic negation, and several saturation instructions (*SAT.B, SAT.BU, SAT.H,* and *SAT.HU*).

To make effective use of instruction forms with a common source and destination register, the register allocation strategy must be appropriately tuned. The dyadic translation for a generic *ADD A, B, C* is:

```
MOV    A, B      ; copy B to A
ADD    A, C      ; A = A+C = B+C
```

For a dyadic instruction form to be usable without the preceding *MOV*, the instruction must represent the last use of the definition residing in the left (or only) operand register, and the compiler must be able to reallocate that register to the result of the instruction. If the result must reside in a different register, then a 32-bit triadic instruction should be used, in preference to a *MOV* followed by a 16-bit dyadic instruction.

The method for implementing register allocation for short form instructions will obviously depend on the general design of the compiler back-end being targeted. In terms of the widely used model of register allocation by coloring an interference graph, it involves constraining the non-interfering result node and the left operand node of a binary operation to have the same color. If the two nodes do interfere, but the operation is commutative, then the right operand node should be checked for interference with the result. If it does not interfere, then swapping the operands will allow the coloring constraint for a dyadic instruction to be applied.

A reordering of operations might be able to remove an interference edge between the result and one of its operands, allowing a short form dyadic instruction to be used. Interference, in general, is dependent on the order of operations. In the canonical order from which the interference graph is typically built, the last use of an input operand might follow the operation, creating interference between the operand and the result. However it might be possible to reorder the operations, such that the current operation follows the other use, and becomes the last use of the operand value in the reordered code. In that case, the interference between result node and the operand can be removed, enabling the nodes to be colored for use of a dyadic instruction.

A contrived example of this type of rescheduling is seen in the following code fragment:

```
extern int e_int_1, e_int_2, e_int_3;
...
void example(void)
{
    int a, b, c;
    ...
    a = b + c;
    e_int_1 = a;
    e_int_2 = b;
    e_int_3 = c;
}
```

Under the canonical order of code generation, both *b* and *c* are live after the add operation, and a 32-bit triadic form of *ADD* must be used to compute *a*. The TriCore assembly code generated for the last four statements might be:

```
    add   d0,d1,d2
    st.w  [a0]%sda_offset(e_int_1),do
    st.w  [a0]%sda_offset(e_int_2),d1
    st.w  [a0]%sda_offset(e_int_3),d2
```

However, the assignment of *b* to *e_int_2* is not dependent on the *ADD*. Promoting that assignment ahead of the *ADD* enables the following generated code:

```
    st.w  [a0]%offset(e_int_2),d;
    add16 d1,d2
    st.w  [a0]%offset(e_int_1),d1;
    st.w  [a0]%offset(e_int_3),d2;
```

1.1.2 16-bit Loads and Stores

The TriCore architecture includes short instruction forms for all of the most frequently used types of Load and Store instructions:

- *LD.BU, LD.H, LD.W, LD.A*
- *ST.B, ST.H, ST.W, ST.A*

Each of these instructions has variants for four different addressing modes:

- Register indirect
- Implicit base
- Implicit destination
- Auto increment

The *implicit base* and *implicit destination* modes each include a 4-bit offset field which is zero-extended and scaled according to the access type. The implicit base register is A15, while the implicit destination register is D15.

The register indirect and the auto increment forms are most useful within loops, for dereferencing pointers and index expressions that have been strength-reduced to pointers, following loop induction analysis. The auto increment form may be used when a dereferenced pointer is post-incremented in the source code. For example:

```
    c = *(p++);
```

Alternatively it may result from back-propagating a separate pointer increment to the point of a preceding use (any isolated pointer increment or decrement should trigger a check for a use with which the operation could be folded).

In some cases it is worthwhile using a 32-bit *LEA* instruction to enable multiple subsequent uses of a register indirect form. However this optimization should be used with discretion, as it requires two uses of the short form to just "buy back" the code space cost of the added *LEA* instruction. If the uses are within a loop however, and there is little pressure on the address registers, it can be worth making the transformation in order to reduce the size of the loop. The assumption is that the reduced loop size will probably yield cycle time savings by reducing instruction cache misses during loop execution. Generally however it is best not to apply the optimization unless it will enable three or more uses of the short instruction form.

1.1.3 Loads and Stores with Implicit Base and Implicit Destination

This section describes how to take advantage of the implicit base and implicit destination load and store forms. These can be used to compactly translate address expressions that reduce to base + 4-bit scaled offset, and can save considerable code space in applications that contain frequent references to fields of pointer-accessed records. However in order to use one of these forms, A15 or D15 must be available at the point required, and the allocation of either register for the instruction must be compatible with any subsequent uses of the data element or address expression.

This means that the data element (in the case of D15) or the address expression (in the case of A15) should have a lifetime that interferes as little as possible with other candidate uses of the register. That usually means a short lifetime. An easy example is a pointer which is loaded and then used to access a record field, after which it has no further uses. A15 can be allocated to receive the pointer, and then be used for an implicit base mode instruction to access the record field. After that access A15 is immediately available for the next candidate use of implicit base mode.

In some cases a pointer will have an extended lifetime, but the individual record fields accessed through the pointer will have short lifetimes. In such cases the implicit destination addressing mode can be used to reduce code space. The record field is loaded into D15 and quickly used, or evaluated into D15 and quickly stored, freeing D15 for its next use.

Because of the special status of D15 and A15, the best way to manage these registers is through a special register allocation pre-phase. Places where the use of one of the registers will save code space should be identified. Uses should be prioritized on the basis of least interference with other candidate uses. This is a code space optimization, and the object is to satisfy the largest number of candidate uses. Following this phase, the two registers become part of the general register pool. Any allocations of the registers made in that context will, by construction, not interfere with the pre-allocated uses. But there is no reason to exclude the registers from the allocation pool if additional opportunities to use them still exist after the pre-allocation phase.

1.1.4 Other Implicit D15 Instructions

In addition to the Load and Store instructions using D15 as an implicit destination or source, there are a number of other instructions that use D15 as an implicit destination or source register. These (described in the following subsections) include:

- **"2.5 Address" Alternatives for Dyadic Instructions**
- **Instructions with 8-bit Constants**
- **Compare Instructions**
- **Compare and Jump**

1.1.4.1 "2.5 Address" Alternatives for Dyadic Instructions

As previously stated, there are 16-bit dyadic forms for the more frequently occurring binary operations. However the dyadic forms can not be used when both input operands are live after the operation. The frequency of additions and subtractions is high enough to justify special "2.5 address" forms for *ADD* and *SUB*. These forms can be used when both input operands are live, or when one input operand is live and the other is a constant. D15 is the implicit destination for these instruction forms.

1.1.4.2 Instructions with 8-bit Constants

D15 is also the implicit destination register for a variant of *MOV* whose source operand is a zero-extended 8-bit constant. It is the implicit source and destination for a variant of *OR* with a zero-extended 8-bit constant as its right operand. The former is useful for storing 8-bit character values to memory, while the latter is intended for setting bits in control variables read from memory.

1.1.4.3 Compare Instructions

Several short compare instructions write the result of the compare to D15. The compares available are *EQ, LT* and *LT.U,* with variants for two register operands, or one register operand and a 4-bit constant. The constant is sign extended for *EQ* and *LT*, and zero extended for *LT.U*.

1.1.4.4 Compare and Jump

Short compare and jump forms, with D15 as an implicit source, are available for *JZ* and *JNZ*, with 8-bit signed offsets, and for *JEQ* and *JNE* with 4-bit unsigned forward offset (in all cases branch offsets are halfword offsets). Variants of *JEQ* and *JNE* exist for the second source operand as either an arbitrary data register or as a 4-bit signed constant.

1.2 Using Complex Instructions

The TriCore architecture features a number of instructions that are more specialized and more powerful than those typically found in a pure RISC architecture. Two obvious examples include the multiply-accumulate (MAC) operations and the bit field extract and insert instructions. The operations that these instructions perform require sequences of two or more instructions with a conventional RISC ISA (Instruction Set Architecture). Taking advantage of them improves both performance and code density.

Depending on the particular compiler technology, there are various approaches that can be taken for recognizing opportunities to use these instructions. The easiest and most natural approach is often to employ a target-specific IR (Internal Representation) rewrite phase ahead of the target-independent optimization phase. The IR rewrite phase uses high-level template matching to find IR subtrees that can be transformed to use the operations. Typically the operations will be expressed in the transformed IR as intrinsic function calls that will be recognized by the code generator.

In other cases it is more appropriate to apply the transformations as “peephole” style optimizations, after the optimization analysis phases.

Regardless of what method the compiler actually uses, some form of pattern matching is involved. The subsections that follow discuss individual instructions or instruction classes in terms of the source code patterns that the operations match.

Not included in this section are the bit instructions (bit load and store, bit move, bit logical and jump on bit). To use these instructions effectively a number of special considerations need to be noted, and they are covered later in a separate section.

1.2.1 Auto-adjust Addressing

Two of the addressing modes available for all 32-bit Load and Store instructions are pre-adjust and post-adjust modes. Both of these modes include a 10-bit signed offset field which is added to the base register, before or after the register is used for Effective Address (EA) generation. In either case the result of the addition updates the contents of the address register that subsequent instructions will see.

The "++" and "- -" operators in C, map naturally into these addressing modes. No special template matching is needed for instances where these operators appear explicitly in pointer dereferences. However TriCore's auto-adjust addressing modes are more general than the C operators in that the size of the adjustment is decoupled from the size of the access type. There will therefore be instances of source code that does not employ those operators, but that can be translated using the auto-adjust modes.

The general pattern for use of the post-adjust mode is:

```
<any dereference (load or store operation) of 'p'>
...
p += <static increment in range, after scaling, [-512, 511]>
```

The ellipsis shown here between the first and second statements implies no intervening use of *p*; its value must have no reached use other than the increment operation. In that case, the increment operation can be hoisted for folding with the Load or Store operation, through use of the post-adjust addressing mode.

For pre-adjust mode, the pattern is the same as above, but in reverse:

```
p += <static increment in range, after scaling, [-512, 511]>
...
<any dereference (load or store operation) of 'p'>
```

Here the increment operation is lowered for folding with the Load or Store operation. All uses of the value of *p* resulting from the increment must be dominated by the dereference operation.

Many opportunities to apply this optimization will not be evident in the initial source code, but will result from strength reduction of index expressions after loop induction analysis. The optimization will therefore be most productive when applied late.

Note that post-adjust addressing is also available for 16-bit Load and Store instructions. For those instructions, the adjustment increment is not explicit, but is implied by the access type.

1.2.2 Circular Addressing

Although the TriCore architecture supports circular buffer addressing for efficient real-time data processing and DSP applications, it is decidedly non-trivial for a compiler to recognize, from generic C source code, when use of circular addressing is appropriate. The problem is that there are many ways that the effect of circular addressing can be expressed in the source code, and it requires specialized and fairly elaborate analysis to be able to recognize them.

It is therefore suggested that the compiler not attempt to recognize implicit opportunities to employ circular addressing. Instead, the compiler should support memory referencing intrinsic functions, that will allow users to explicitly employ circular addressing.

The intrinsics will require a data type for a circular buffer pointer. The required data type can be implemented through a C language extension, or as a C++ class.

1.2.3 Compare and Jump

```
    if (e1 relop e2) {
```

The usual RISC translation for this statement is a relational compare followed by a conditional jump on a zero / non-zero compare result. For TriCore however, the compare and jump can usually be folded.

The folding may require some rewriting of the expression. The available forms are *JEQ, JNE, JLT, JGE, JLT.U* and *JGE.U,* with options for the right operand as either a general data register or a four-bit constant (the constant is sign extended for the first four and unsigned for *JLT.U* and *JGE.U*).

A branch on "<=" for example, can be accommodated by the standard tricks of swapping the left and right register operands and using *JGE*, or by adding one to the constant right operand and using *JLT*.

1.2.4 Multiply-Add

A multiply followed by an add can be replaced by a multiply-add (*MADD* instruction) whenever the result of the multiply has no uses other than the add. When not compiling in debug mode ("-g"), that should include instances where the result of the multiply is assigned to a local variable, but where that definition of the variable has no uses other than the add.

1.2.5 Absolute Value

Although *stdlib.h* includes the absolute value function *abs()*, it is frequently implemented with a preprocessor macro, of the form:

```
    #define ABS(A) ((A) < 0? -(A): (A))
```

This source construct (and equivalent variations) should be translated using the TriCore *ABS* instruction.

Replacing an actual call to the *abs()* library function with the *ABS* instruction is problematic. In theory the user should be able to shadow the standard library with an *abs()* function that does something different than (or in addition to) the expected. It might for example create a histogram of the input arguments for analysis purposes. If the compiler helpfully replaces the call with the inline instruction, it will be violating ANSI rules. The replacement, if done at all, must be done at link time, when it is known that the *abs()* function being called is really the standard library function.

1.2.6 Min and Max

TriCore has *MIN* and *MAX* instructions, with the obvious definitions. The compiler should recognize the frequently used macro implementations for these operations (below) and replace them with calls to intrinsic functions for the MIN and MAX instructions.

```
#define MIN(A,B) ((A) < (B) ? (A) : (B))
#define MAX(A,B) ((A) > (B) ? (A) : (B))
if(a < b) c = a; else c = b;
if(a > b) c = a; else c = b;
```

The semantically equivalent code using the >= or <= operators should also be identified.

In evaluating the macro arguments, side effects must of course be preserved. Reduction of the templates given above to single *MIN* or *MAX* instructions, requires that the expressions for "A" and "B" both be free of side effects. If either expression has side effects then the macro implementations shown above for *MIN* and *MAX* may represent programming errors, because they evaluate one of the expressions a second time. However that is the user's problem - the compiler must still produce a translation that is semantically consistent with the source code. There are of course more rigorous macro definitions that evaluate the argument expressions only once. However, in terms of the pattern matching templates for the *MIN* and *MAX* instructions, there is no difference.

1.2.7 Count Leading (CL)

The TriCore instruction set includes instructions to count the number of leading zeroes (*CLZ*), ones (*CLO*) or redundant sign bits (*CLS*). These are useful for supporting software floating point and fixed point math, where precision is explicitly managed. They are also useful for dispatching from bit-mapped priority queues, and for set traversal.

There is no simple way to express these operations in C, so it is not possible to use template matching to find opportunities to use them. For the programmer to take advantage of these instructions, they must be supported as intrinsic functions.

1.2.8 Conditional Instructions

There are a limited number of conditional instructions in the TriCore instruction set. They include conditional move (*CMOV* and *CMOVN*), select (*SEL* and *SELN*), conditional add (*CADD* and *CADDN*) and conditional subtract (*CSUB* and *CSUBN*). The control condition is the true (non-zero) or false (zero) status of a specified data register (The '*op*N' forms perform *op* when the control condition is false).

Appropriate use of these conditional instructions can save both code space and execution time. The avoidance of branching is of greater importance in code generated for TriCore 2.0 than TriCore 1.2/1.3 because of its deeper pipeline. However recognizing when the use is "appropriate" is not always obvious.

The *SEL* instruction is the natural translation for the C "?" operator, whenever the alternative expressions are "simple" and free of side effects.

```
<local> = <condition> ? <expr_1> : <expr_2>;
```

This statement can be translated by evaluating <condition> into register Dd, <expr_1> into Da, and <expr_2> into Db, and issuing the instruction:

```
sel Dc,Dd,Da,Db
```

Dc is the register allocated to *<local>.*

This is usually both faster and more compact than the *if...else* implementation of a conditional jump around an assignment and unconditional jump around the else part. However *<expr_1>* and *<expr_2>* must be free of side effects, because the semantic definition of the "?" operator specifies that only one of the two expressions is to be evaluated. Also, if one or both expressions are complex and costly to evaluate, the "if. else" implementation may be more run-time efficient.

The most obvious application for the conditional move and conditional add or subtract instructions is in the translation of very simple *if* statements. For example:

```
if (<condition>) <local_a> = <local_b>;
```

This should be translated with a single conditional move from the register allocated to *<local_b>* to the register allocated to *<local_a>*, gated by the register allocated to *<condition>*. Similarly:

```
if (<condition>) <local_a> += <local_b>;
```

This should be translated with a conditional add.

In both of these cases the availability of normal and complement forms ('*op*N') of the conditional instructions can potentially simplify the evaluation of *<condition>*. The obvious case is when the root of the expression tree for *<condition>* is the logical complement operator. The complement operation can be avoided by selecting the complement form of the conditional operation. Less obvious cases are when the expression tree for *<condition>* can be simplified by complementing the condition. A frequent example will be when *<condition>* is simply *"<expr> == 0"*. This is equivalent to

"!(<expr>)", and allows *<expr>* to be used directly as the control expression for the complement conditional form.

1.2.9 Extending the Use of Conditional Instructions

Although the TriCore architecture does not support the conditional execution of most instructions, it is frequently possible to combine one or more unconditional instructions, followed by a *SEL* or a *CMOV* to avoid branching. The instructions executed unconditionally in a conditional context (i.e. when literal adherence to the control flow path would branch around them), must be free of adverse side effects. In general that means they cannot trap, and will develop their results in a General Purpose Register (GPR) that can be conditionally used.

One of the more complex examples of this is seen in the emulation of conditional stores. If the left hand side of a conditional assignment statement is a global variable or pointer dereference rather than a local, then the required conditional operation is a store instruction. The TriCore instruction set does not include any conditional stores. However, as explained below, it may still be possible to rewrite the Internal Representation (IR) to emulate the effect of a conditional store, and avoid branching.

In general compilers should always handle non-volatile global variables by "localizing" the global variable references. Uses of the global variable are replaced by uses of a local variable that is initialized to the global variable at a point that dominates the uses. Definitions of the global variable are replaced by definitions of the local variable, coupled with a copy to the global variable from the local at a point that post-dominates the definition, with respect to the function's exit node, or at any point such as a call to an un-analyzed external function, where the value of the global variable may be used. The same "localization" process can be applied to pointer dereferences, within regions where the dereferenced memory is provably safe from aliasing.

After localization, it is often possible to use conditional moves or adds to translate conditional assignment statements. The value that would have been conditionally stored to the global variable is, instead, conditionally moved to the local surrogate for the global variable. Eventually the local surrogate will be unconditionally stored back to the global variable.

Another example of using an unconditional operation followed by a conditional move to extend the use of conditional instructions is in handling the general case for a conditional add. Note that TriCore's definition of conditional add does not allow it to be used in translating the following:

```
    if (<condition>) <local_a> = <local_b> + <local_c>;
```

TriCore defines "`cadd dc,dd,da,db`" as:

```
    dc = dd ? da + db : da
```

rather than:

```
    dc = dd ? da + db : dc
```

The latter would arguably be a more intuitive definition, but its implementation would require either four parallel operand reads in one cycle, or a true "conditional write" to the destination register. Neither solution is really practical. The problem with the former is obvious. The problem with the latter is that it would severely complicate register forwarding and restrict future high performance implementations. As a result, the conditional add will normally be used only in cases where one of the add source operands is also the destination operand;i.e. the C "+=" operation.

However, the example above can be translated by combining an unconditional add with a conditional move. Even though the add is executed at times when it is not needed, the *ADD / CMOV* sequence is cheaper than a conditional jump over an unconditional *ADD*.

1.2.10 Accumulating Compare Instructions

The TriCore's 3-input accumulating compare instructions are powerful operations for evaluation of complex logical expressions. Take the following C expression for example:

```
((e1 < e2) || (e3 > e4))
```

This can be translated with just two instructions (assuming e1 through e4 are free of side effects, and already reside in or have been evaluated into registers):

```
lt16  d15,d1,d2
or.lt d15,d4,d3
```

The more complex the expression, the greater the payoff of the 3-input instructions. The general pattern is to employ a regular compare or logical instruction to evaluate the first intermediate result (*"e1 < e2"*, in the example above) into a register. That register serves thereafter as an accumulator for the final expression result. A sequence of one or more 3-input instructions then follow, each using the accumulator register as the 3rd input and destination. Each of the 3-input instructions replaces two regular instructions:

- One to evaluate the sub expression into a register
- One to merge that result into the accumulator

Some rewriting of the expression tree may be required in order to take maximum advantage of these instructions. For example, in the absence of hardware floating point relational compare instructions, a C macro to perform a "less than" compare on two single precision floating point numbers, using only integer operations, can be written as:

```
#define LT_F(A,B)(((A<B) ^ ((A<0) && (B<0))) && (((A|B)<<1)!=0))
```

Note: This assumes that the macro arguments supplied for A and B are simple variables, that have been declared as integers. A robust macro definition would be substantially longer and harder to read.

The macro relies on the fact that, under IEEE 754 format, a floating point relational compare is the same as an integer compare, except when both numbers are negative or when the two numbers are plus zero and minus zero. When both numbers are negative, the result is inverted. Plus and minus zero are forced to compare as equal.

The optimal TriCore code to translate this rather complex macro is only six instructions. Assuming *A* is in D4, *B* is in D5, and the result is returned in D2, the assembly code would be:

```
LT       D2,D4,0
AND.LT   D2,D5,0
XOR.LT   D2,D4,D5
OR       D3,D4,D5
SH       D3,1
AND.NE   D2,D3,0
```

To get this code, the expression has been evaluated as if it had been written:

```
((((A<0) && (B<0)) ^ (A<B)) && ((A|B) << 1) != 0)
```

The translation would not have been as efficient if *A < B* had been evaluated first, because it would then have been the left-hand operand of the *XOR* operation. The right-hand operand would have been the *AND* of *(A<0)* and *(B<0)*. TriCore does not have an *XOR.AND* instruction, so a compound instruction could not have been used. The reordering of the operations, made possible because XOR is commutative, allows the*XOR.LT* operation to replace separate *LT* and *XOR* operations.

1.3 Using Bit Instructions

The TriCore architecture features a powerful set of single bit and bit field instructions. These were designed particularly for use in control applications coded in assembly language, where their use can reduce both code size and execution time. However they are also useful in translating certain types of C code.

There are three contexts in which the bit and bit field instructions can be applied for C translation:

- For direct high level access to named fields in packed records
- For direct access to discrete boolean variables that the compiler has elected to pack into a single word variable held in a register
- For template-based reduction of low-level bit and bit field operations expressed in C using word-logical operations ('&', '|' and '^'), literal constants and shift operations.

For access to bit fields in packed records, TriCore's *EXTR* and *INSERT* instructions can be used reasonably directly with no great complications, provided that the compiler's front-end stages have not already expanded the references at the Internal Representation (IR) level, for a RISC ISA model that lacks bit field operations. For example: "*rec.field*" ==> "*(rec & field_mask) >> field_offset*". A code generator for TriCore would prefer to have the reference left as "*rec.field*", with linked symbol table information that would supply the parameters for an *EXTR* or *INSERT* instruction.

It is not common for compilers to collect discrete boolean variables and pack them into a single temporary that can be held in a register. However for TriCore that can be a very useful optimization. TriCore has the necessary instructions to set, clear or toggle individual bits within a word, to perform logical operations on individual bits, and to conditionally branch on individual bit status. Packing a large number of discrete boolean variables into a single word held in a register can be particularly helpful for code that implements complex finite state machines for control applications.

Since the C language does not feature standard intrinsic functions for accessing bits and bit fields, a great deal of code gets written using preprocessor macros that express these functions in terms of operations that *are* available in standard C: logical operations on integer variable and literals, combined with shift operations. When input is expressed in that form at the IR level, there is no way for a low-level code generator to recognize the opportunity to exploit a bit field instruction. It is therefore necessary to employ a template-matching phase on the IR that can recognize these patterns, and replace them with intrinsic function calls. The intrinsic function calls then map into the appropriate bit and bit field instructions.

Specific template-matching optimizations that can be applied are discussed in the subsections that follow. All of these optimizations involve detection of particular forms of literal operands. There are three forms of interest:

1. Single-bit masks, whose value is 2^n
   – *n* is the position of the bit
2. Right-aligned field masks, whose value is "2^w - 1"
   – *w* is the width of the field
3. More general field masks, whose value is "(2^w - 1) << p"
   – *w* is the width of the field and *p* is the position of its right-most bit

Literal values encountered in the template-matching pass must be tested for these cases. The test to see if a value *m* is a single-bit mask, is:

```
((m != 0) && ((m - 1) & m) == 0)
```

The test for a right-aligned field mask is:

```
((m != 0) && ((m + 1) & m) == 0)
```

The test for a field mask that may or may not be right-aligned is:

```
((m != 0) && (p = ((m - 1) | m), ((p + 1) & p) == 0)
```

1.3.1 Template Matching for Extract

```
EXTR.U  Dc,Da,p,w
```

The TriCore instruction given here can be expressed in C as either:

```
    Dc = (Da >> p) & (2^w-1);
```

or

```
    Dc = (Da & ((2^w-1) << p)) >> p;
```

In the first case the source operand (Da) is right shifted before *AND*-ing with a *w*-bit mask. In the second case the operand is *AND*-ed with the mask before shifting.

An IR expression whose root operator is '&', with left or right operand sub expression a literal of either form; i.e. `(2^w-1)` or `((2^w-1) << p))`, can be replaced by an intrinsic function call for the *EXTR.U* instruction. If the other operand sub expression has a root operator of >>, then the shift amount for that sub expression can be added to the *p* argument of the intrinsic function call, and the >> operator eliminated.

If the & operator was the left operand to a >> operator, then the shift amount for the >> operand can be subtracted from the *w* argument of the intrinsic function call, and the >> operator eliminated.

The unsigned *EXTR.U* can be converted into a signed extract *EXTR*, if the result of the *EXTR.U* is found to be sign extended. Sign extension of a *w*-bit field *f*, may be expressed in C as either of the following:

```
(f << (32-w)) >> (32-w)
```

This relies on the common (but not guaranteed) use of arithmetic shifts for the ">>" operator. More correctly, it may be expressed as:

```
(f >= 2^(w-2) ? f - 2^(w-1) : f
```

(or by equivalent variations.)

1.3.2 Template Matching for Insert

```
INSERT Dc, Da, Db, p, w
```

This TriCore expression can be expressed in C as either:

```
m = (2^w - 1) << p;
Dc = (Da & (~m)) | ((Db << p) & m);
```

or

```
m = 2^w - 1;
Dc = (Da & ((~m)<< p)) | ((Db & m) << p);
```

There are variations for Db as either an expression evaluated into a register, or as a static constant. For extract, the compiler should look for constants of the form:

```
(2^w-1)
```

or

```
((2^w-1) << p)
```

as operands to '&'. In this case it should also look for the logical complements of such constants and attempt to match them to one of the *INSERT* templates.

1.3.3 Bit-Logical Instructions

The bit-logical instructions include the simple bit-logical instructions and the so-called accumulating bit-logical instructions. They are mainly intended to facilitate programming of finite state machines for control applications.

The simple bit-logical instructions (*AND.T, NAND.T, OR.T, NOR.T, XOR.T, XNOR.T, ANDN.T* and *ORN.T*) take two bit values as input operands, and return a TRUE / FALSE (1 / 0) result in the destination register. An input bit can be any statically specified bit from any data register.

Occasionally C source code will yield expressions that can be reduced to one of these operations, through template matching. An example for AND.T would be:

```
    (<expr_1> & 0x02) && (<expr_2> & 0x08)
```

which might be translated as:

```
    ..                      ; evaluate <expr_1> into D4
    ..                      ; evaluate <expr_2> into D5(1)
    AND.T D2,D4:1,D5:3      ; evaluate result into D2
```

More frequently however, opportunities to use these instructions will result from allocation of boolean variables to individual bits of one or more compiler-allocated "collected boolean variable" words.

For example, a program might have several local flag variables, *f1, f2, ... fn*. The compiler, after recognizing that the variables are in fact boolean variables, can allocate register D0 to hold all of them. The individual variables would be allocated to bits 0 through *n*-1 of register D0. From then on, logical functions on those variables can be translated using bit logical instructions operating on the corresponding bits within D0. For example:

```
    if (f1 || (f2 && f3)) ..
```

This statement could be translated as:

```
    AND.T D2, D0:1, D0:2
    OR.T  D2, D2:0, D0:0
    JZ    D2, .else_part
    ..
.else_part:
```

Note that the *OR.T* instruction references bit zero of register D2, where the result of the preceding *AND.T* is held.

To save memory space, boolean variables that are static or global (and not *volatile*) can also be collected into individual words in memory. When a boolean variable of that type is used, the word containing it must be loaded into a register. The variable can then be accessed as an input operand by any of the bit logical instructions.

1) Assumes that <expr_2> has no side effects. The C semantics for "&&" require that the right hand operand not be evaluated when the left hand operand is FALSE. But if the expression for the right hand operand has no side effects, it is safe to evaluate it anyway.

To update the register copy of such a variable, it is normally necessary to use the *INS.T* instruction to move the source bit value into the target bit location. To update the memory copy, the register copy of the word containing the variable can be stored directly, provided that it still holds currently valid copies of all the other variables in the word; i.e. from the point where the word was loaded to the point where it is being stored, there have been no function calls or pointer dereferences that might have updated one of the variables contained in the word. If that condition does not apply then the *LDMST* instruction must be used to update only the target bit within the memory word.

1.4 Call Related Optimizations

By defining a hardware call-return model that automatically saves and restores the full set of upper context registers, the TriCore Instruction Set Architecture (ISA) eliminates the need for nearly all function prolog and epilog code. This contributes to high code density. However, if TriCore did not provide a mechanism to save and restore upper context registers quickly, automatic context saves and restores would actually hurt performance. With a good optimizing compiler, the average number of registers that need to be saved and restored is much less than a full upper context.

TriCore employs a combination of methods to achieve fast context saves and restores. Both TriCore 1.3 and TriCore 2.0 implement dual banks of upper context registers. For an average of roughly 70 - 80% of calls and returns, the context save or restore can be accomplished simply by toggling the upper context bank selector. For the remaining cases in which it is necessary to save or restore values in the alternate bank to or from memory before switching to it, a wide path to memory is employed, that can transfer multiple registers per cycle. For TriCore 1.3 that path is 128-bits wide, allowing four registers to be transferred in parallel. For TriCore 2.0 the path has been reduced to 64-bits to provide a more "generic" memory interface. As a result, the 20 - 30% of calls and returns for which the context save or restore cannot be accomplished by a bank toggle, require nine cycles to complete. This creates a need for optimizations to reduce this overhead.

1.4.1 FCALL (Fast Call) Instruction

To reduce the performance impact of the narrower memory interface in TriCore 2.0, the *FCALL* instruction was added to the instruction set. An *FCALL* (Fast Call) is similar to a regular *CALL*, except that it preserves only one quarter (four words) of the upper context. The four preserved words are the PCXI (Previous Context Information) register, the PSW (Processor Status Word) and the A10-A11 register pair, caller's SP (Stack Pointer) and RA (Return Address).

A new "F" flag in the post-call PCXI register records whether the call was a regular or fast call. It qualifies the operation of a subsequent *CALL* or *RET*. For a *CALL* or *FCALL*, it specifies how much of the context held in the alternate bank of UCX registers must first be saved to memory before the called function can start using that bank.

For a *RET (Return from CALL)*, it specifies how much of the alternate bank must be reloaded from its CSA (Context Save Area) memory before switching banks and returning to the caller. A quarter context save or restore takes only three cycles rather than nine, in those cases where an actual memory transfer is required.

The compiler can and should replace a regular *CALL* with an *FCALL*, whenever there are no values in any of the upper context registers (other than A10 - A11) that are live across the call. The called function has no code dependence on whether it was called with a regular *CALL* of an *FCALL*, so the choice between *FCALL* and *CALL* is strictly a matter of conditions in the calling function.

The compiler can convert *any* regular *CALL* to an *FCALL*, by inserting explicit caller-side saves of any values that are live across the call. Any use of those values after the call will require the saved values to be loaded. There is of course a cost in code space for every explicitly saved and restored value. However, if the number of registers involved is small (typically three or less), the conversion can "pay-off" in execution performance when the called function in turn, makes a call. The reward is greatest when the code surrounding the call is dominated by integer pipeline operations. In that case, the saves and restores may be scheduled into Load/Store (LS) pipeline slots that would otherwise go unused.

1.4.2 Tail Call Conversion

Even more productive than conversion of *CALL*s to *FCALL*s is the conversion of calls to jumps. This can take place whenever a call is followed immediately by a return, or by a jump to a return. This is termed *"tail call conversion"*. It subsumes, as one particular case, elimination of tail recursion.

Tail call conversion eliminates the implicit save of the upper context registers that is associated with the *CALL* instruction. While the context save is usually transparent, avoiding it when possible is still a good policy. In doing so it may reduce the amount of system memory needed for Context Save Areas (CSAs), and it avoids the timing uncertainty introduced by the possibility of a cache miss on the context save, or by added cache misses later due to displacement of cache lines by the Context Save Area.

One consequence of tail call conversion that must be taken into account is that it effectively removes the caller from the call chain, and makes it appear as if the called function had been called by the immediate caller's caller. That does not affect the semantics of program execution, but it may confuse a debugger. The optimization should probably be suppressed when the "-g" compiler option is specified.

A further refinement of tail call conversion is to increment the Stack Pointer (SP) by the frame size of the calling function before the jump, provided that the calling function has no locals with exported addresses that might be referenced from the function jumped to (the most obvious case would be a local variable or structure passed by address to the function in question). This optimization helps to limit stack growth, especially when making tail calls to recursive functions. Again however, it complicates debugging, and should probably be suppressed when the user compiles with the "-g" flag.

1.5 QSEED FPU (Floating Point Unit) Instruction

TriCore 1.2/1.3 can be additionally configured with a single precision Floating Point Unit (FPU) via a co-processor interface that enables significant acceleration of single precision floating point operations over that of emulation libraries. A functionally equivalent FPU has been integrated into the pipeline TriCore 2.0, to enable even more efficient execution of FPU instructions (see Chapter 2 for differences in TriCore 1.2/1.3 and TriCore 2.0 pipelines).

The *QSEED.F* FPU instruction allows a more efficient implementation of a single precision floating point square root function. It returns an approximation to 1.0f divided by the square root of the supplied floating-point value. The implementation guarantees that the difference between the infinite precision result and the returned estimate shall be less than $2^{-6.75}$ times the infinite precision result. The returned value can then be used as a starting point for an efficient software implementation of a square root function using Newton-Raphson refinement iterations (see equation below).

To find the square root of *a*:

Iterate, where: $x_n \approx \frac{1}{\sqrt{a}}$ (i.e. $x_n =$ the result of *QSEED.F*):

$$x_{n+1} = \frac{3}{2}x_n - \frac{1}{2}ax_n^3 \rightarrow \frac{1}{\sqrt{a}}$$

Then:

$$\sqrt{a} = \frac{a}{\sqrt{a}}$$

1.6 Miscellaneous Considerations

1.6.1 LOOP Instructions

The instruction *LOOP*, provides zero-overhead looping for counted loops. Placed at the end of the loop it's first execution initializes a loop cache entry. From then on the loop cache entry monitors the instruction fetch address, and executes the loop instruction when it sees its address coming up in the fetch address stream.

In TriCore 1.2/1.3 and TriCore 2.0 implementations there are only two loop cache entries. It is legal, but counter-productive, to use *LOOP* for more than an inner loop and an immediately enclosing loop. If an outer loop includes two or more inner loops within its body, it is counter-productive to use *LOOP* for the outer loop. The *LOOP* instruction in the outer loop will always be executed after the instructions in the inner loops, and will never find a loop cache entry set up for its own PC (Program Counter) and target address combination. Every execution will appear to be a "first" execution - one cycle more expensive than a regular conditional branch. Furthermore, displacement of one of the loop cache entries for the inner loops by the *LOOP* instruction in the outer loop will cost an extra two cycles for re-creation of the loop cache entry for the next iteration of that inner loop.

A variant of the *LOOP* instruction *LOOP.U*, can be used for zero-overhead *while* loops. It is an unconditional *LOOP* instruction, with no loop counter operand. Placed at the bottom of the loop, *LOOP.U* seamlessly stitches the end of the loop to the top. It consumes one loop cache entry, and is therefore subject to the same payoff considerations as the regular *LOOP* instruction.

1.6.2 Literals

TriCore implementations include physically separate instruction and data scratchpad memory, and physically separate instruction and data caches. There is a substantial runtime cost to read data values from the code scratchpad memory, or to execute code from the data scratchpad. The latter is not usually an issue since code is normally placed automatically in code sections that get located to the code memory. But some compilers embed string constants and other literals within the code sections, as the "natural" place for read-only values.

That strategy is inappropriate for TriCore configurations with separate instruction and data memory. In fact the TriCore architecture deliberately provides no PC-relative data addressing mode. For TriCore, literal values should be placed in a read-only data section separate from the code section.

For 32-bit literals in general, and for address literals in particular, the compiler has a choice of implementation strategies. It can embed them in the instruction stream, as the two 16-bit immediate operands of a *load high / add immediate* instruction pair. Alternatively it can load them from the read-only data section. The former implementation strategy incurs a code space penalty of one word for every load of a given literal, after the first load, but leads to more predictable execution times for system configurations involving cached DRAM. The latter strategy is more efficient for configurations with SRAM data memory. It also has the virtue of allowing address literals to be grouped together in memory, to simplify debugging and facilitate dynamic linking. To enable single-word loads from the address literal pool however, the literal pool must be merged with the "small data area" addressable through dedicated base register A0.

1.6.3 Shift Instructions

TriCore's shift instructions are slightly unusual in that they are "generic" shifts, taking a signed shift count. A positive count shifts left, and a negative count shifts right. For static shift values there is no complication, as the compiler simply emits a positive or a negative shift count according to whether the operator being translated is "<<"or ">>".

For dynamic right shifts, the shift count must be negated. To allow recognition of common sub-expressions and other optimizations, the negation should be done at the IR (Internal Representation) level, *not* in the low level code generation template expansion for the right shift operator.

1.6.4 Switch Statements

It would be possible to write a separate book about code generation for switch statements and the trade-offs that various strategies involve for particular architectures. Here however, the discussion focusses on some issues that are particularly relevant for TriCore.

For small switch statements, with a small number of arms and a small range of switch values, the most compact implementation is a straightforward series of short *JNE* (Jump if Not Equal) instructions from the beginning of each arm to the next. The switch expression must be evaluated into D15, and the non-default values for the expression must lie within the range [-8, +7]. Then if each arm of the switch statement occupies less than 16 halfwords, all of the *JNE* instructions can use 16-bit forms. The net cost for the switch statement is zero setup, and one halfword per non-default arm.

The following example illustrates this strategy:

```
enum color {red, green, blue, other};
...
switch (c) {
case red:
    ...
    break;
case green:
    ...
    break;
case blue:
        ...
        break;
    default:
        ...
}
```

The translated code, using the compact code strategy described, would be:

```
    <evaluate (c) into d15>
.red_test:
    jne16 d15,0,.green_test
    ...             ; code for 'case red'
    j .continue
.green_test:
    jne16  d15,1,.blue_test
    ...             ; code for 'case green
    j  .continue
.blue_test:
    jne16  d15,2,.green_test
    ...             ; code for 'case blue
    j  .continue
.default:
    ...             ; code for default case
.continue:
```

Note that this strategy results in one conditional jump taken for every unsuccessful switch value test. If the compiler has access to profiling data, or some other means to determine the execution frequency of each arm, it can order the arms to minimize the number of unsuccessful tests and branches. Otherwise it may prefer to group the tests together, so that at most one branch ends up being taken. The code for the example above would then become:

```
    <evaluate (c) into d15>
    jeq  d15,0,.red_case
    jeq  d15,1,.green_case
    jeq  d15,2,.blue_case
.default:
...                     ; code for default case
    j   .continue
.red_case:
    ...                 ; code for 'case red'
    j   .continue
.green_case:
    ...                 ; code for 'case green'
    j   .continue
.blue_case:
    ...                 ; code for 'case blue'
.continue:
```

Note the code movement for the default case, allowing fall-through from the last case test.

With this approach the branches have greater spans, and are less likely to resolve to 16-bit forms. The approach therefore involves a degree of space / time trade-off.

For switch statements with many arms, neither of the above approaches are particularly attractive. Direct indexing of some form of jump table by the normalized case expression will avoid a long series of conditional jumps, at the expense of some initial overhead to access the table entry. The most straightforward and widely implemented approach uses a table of 32-bit addresses of arm labels. Beyond the recommendation to put the table in read-only data memory, rather than code memory, this approach looks and works the same for TriCore as for any standard RISC architecture. However, the memory overhead of one word per normalized switch expression value can be annoying, particularly for switch statements which have many case labels for each distinct switch arm.

The memory overhead per switch expression value can often be reduced to just one byte or one halfword by using a table of relative offsets, rather than a table of addresses. A dummy Jump and Link instruction (*JL*) can be used to create a base address, to which the offset loaded from the jump table is added, before issuing an indirect jump to the appropriate switch arm.

A limitation of this approach is that the compiler must know something about the range of offset values that the table will need to hold. It can leave it to the Linker to resolve the specific offset values, but the compiler must allocate table entries that are wide enough to hold the offset values. If the compiler does not try to support 8-bit offset tables and limits itself to 16-bit offset tables, it can almost always assume that 16-bits will be a sufficient offset. In the extremely rare case of a single switch statement with code spanning more than 128KB, the insufficient width would be detected by the Linker. The compiler would need a switch to force the use of full 32-bit addresses in the switch table.

A two-level lookup is another alternative for saving table space when handling switch statements with an average of many case labels for each arm. The compiler knows exactly how many arms there are, and how many index bits are needed to address that many entries in a table. Assuming the number of arms is 256 or less, an 8-bit scaled index is sufficient to index the address table. So the compiler builds a table of 8-bit index values, which in turn is indexed by the normalized case expression value. The index value loaded is then used to access a short table of 32-bit switch arm addresses.

2 Implementation Information

2.1 Pipeline Model and Instruction Scheduling

This section provides an overview of the pipeline models for both the TriCore 1.2/1.3 and the TriCore 2.0 implementations of the architecture. The purpose is to enable compiler writers to construct models sufficient for use by an instruction scheduler.

Note: Implementations of the TriCore architecture before 1.2 were not commercially available and are therefore not discussed here. The Tricore 1.2 and 1.3 (TriCore 1.2/1.3) implementations are sufficiently similar to be described together. Differences with TriCore 2.0 are clearly noted.

Instruction scheduling aims to improve performance by:

- Avoiding pipeline stalls as much as possible
- Taking advantage of the architecture's parallel issue capabilities

Instruction scheduling does not require the level of detail in the pipeline model that would be needed to construct a cycle-accurate instruction simulator for example. In principle, only details that might affect scheduling decisions are relevant. The fact that a stall cycle will occur under particular circumstances is theoretically of no interest to the compiler, if there is nothing the scheduler can do to avoid it.

In practice it is difficult to construct a model that captures *only* the information relevant to instruction scheduling. The models documented here undoubtedly contain more detail than strictly needed for that purpose. However the point is that they are not specifications for the actual pipeline implementations, and are not intended for constructing totally cycle-accurate instruction simulators. An instruction simulator that integrates the pipeline models described here will be very close to cycle-accurate, assuming that it also integrates accurate cache and memory models. However it would not be 100% accurate in all cases.

2.1.1 First Approximation

At least 90% of the performance improvement achievable through instruction scheduling can normally be achieved with a fairly crude model of the pipeline. As a first approximation, the model described below serves for both TriCore 1.2/1.3 and TriCore 2.0.

2.1.1.1 Simple Pipeline Model for TriCore 1.2/1.3 and TriCore 2.0

TriCore 1.2/1.3 and TriCore 2.0 implementations have two regular instruction pipelines, together with a special zero-overhead loop instruction cache. The two regular pipelines are referred to as the *Integer pipeline*, and the *Load-Store (LS)* pipeline.

In terms of instruction issue characteristics, there are four categories of TriCore instructions:

1. Integer pipeline instructions
2. LS (Load/Store) pipeline instructions
3. Dual pipeline instructions
4. The zero overhead loop instructions

The Integer pipeline instructions are the arithmetic and logical operations on the data registers. The LS pipeline instructions are the load and store instructions proper, plus arithmetic operations and moves on the address registers. Also included are unconditional jumps and conditional jumps on address register tests. The dual pipeline instructions are those that effectively issue to both pipelines in parallel[1)].

A common instruction fetch unit feeds both pipelines. The fetch unit is capable of issuing, in one cycle, an integer pipeline instruction to the integer pipeline and an immediately following LS instruction to the LS pipeline.

Because of the split address and data register files, load instructions can inherently have no input dependence on any integer pipeline instruction. Store instructions can have an input dependence on a preceding integer operation that defines the value stored. However, under the current pipeline implementations, stores occur at a late stage of the pipeline. As a result, a store is able to issue in parallel with an integer pipeline instruction that defines the source value for the store.

A write-after-write output dependence is theoretically possible when a Load instruction follows an integer instruction, and loads the same register that was the destination for the integer instruction. However that would render the integer instruction dead, as far as the C execution model is concerned, and there would be no reason for the compiler to have emitted it. Therefore, for the compiler's purposes, to a first approximation, any LS pipeline instruction can issue in parallel behind any integer pipeline instruction.

An integer pipeline instruction cannot issue in parallel behind an LS pipeline instruction. There is no “cross-over” path in the instruction fetch unit to allow for that. Therefore, if we use “D” to represent an integer (Data) pipeline instruction, and “A” to represent a Load-Store (Address) pipeline instruction, the sequence “DADA” will issue in two cycles, whereas “DAAD” will require three cycles.

1) In some cases, such as ***ADDSC.A***, the instruction requires resources from both pipelines in order to execute. The instruction can actually be dual issued but will stall the other instruction in order to monopolize the resources. In other cases there is no inherent requirement for resources from both pipelines, but suppression of dual issue avoids complications and simplifies the implementation. That applies for example, to conditional jumps on data register tests.

Dual pipeline instructions cannot issue in parallel with either integer pipeline or LS pipeline instructions. There are relatively few instructions in this category. *ADDSC.A, MOV.A, MOV.D*, and the address compare instructions (*EQ.A,* etc.) are the only ALU-style instructions. Conditional jumps on data registers, and the *CALL* and *RET* instructions are the other main instances[1].

A zero-overhead loop instruction will issue in parallel behind any of the other three categories of instruction, provided that it is resident in a loop cache entry. To be resident in a loop cache entry it must have been previously executed, and must not have been evicted from its loop cache entry by the execution of another *LOOP* instruction.

2.1.1.2 Simple Scheduling Strategy

Under the model described above, the optimum instruction scheduling strategy within a basic block is simply to interleave Integer pipeline (IP) and LS pipeline instructions as much as possible. Therefore an optimal instruction selection strategy will result in a 1:1 mix of LS to IP instructions. A number of the operations that can be performed with IP instructions can also be performed by LS instructions using the address registers. A successful instruction selection strategy will then use Address registers as loop counters (even if not using the LOOP instruction), for simple arithmetic, conditional branching and address manipulation. A useful side effect of this strategy is to reduce pressure on the Data registers and therefore reduce spilling of variables to the stack.

If a dual pipeline instruction must be scheduled, it should preferably follow an LS pipeline instruction, or another dual pipeline instruction. There is no parallel issue slot following such instructions, so scheduling a dual pipeline instruction there will not waste the potential parallel issue slot that follows an integer pipeline instruction. The following table explains this strategy. It gives the priority for selection of the next instruction (column headings), as a function of the pipeline category of the preceding instruction (row headings).

Table 1 Instruction Scheduling Priorities

Preceding Instruction	Next Instruction		
	Integer	Load / Store	Dual
(Block Start)	2	3	1
Integer	3	1	2
Load / Store	2	3	1
Dual	2	3	1

[1] It is possible to remove conditional jumps on data registers from this category, and this may be a future performance enhancement. It requires that any LS pipeline instruction that issues in parallel with it be "dismissable", in the event that the branch is taken.

The relatively high priority given to selection of dual pipeline instructions is intended to retire them early, when there is no penalty, so that they unblock dependent LS instructions and enable more pairing of integer and LS pipeline instructions. In practice they will have little impact on scheduling. The only dual pipeline instructions that a compiler will normally generate are:

- Scaled index addition, *ADDSC.A*
- The cross-file move instructions *MOV.A* and *MOV.D*
- The address comparison instructions (*EQ.A*, etc.)
- Conditional jumps on data registers
- The *CALL* and *RET* instructions

Note: The system and context switch instructions are also implemented as dual pipeline instructions, but they are not generated by the compiler, except in response to special intrinsic function calls. In the latter case, the instruction scheduler is not allowed to move any code across the instructions.

Of the dual pipeline instructions generated, only *ADDSC.A, MOV.A, MOV.D*, and the address comparison instructions typically present much latitude for scheduling. Branch instructions are common of course, but are not subject to scheduling. By definition, a branch terminates a basic block and must follow all other instructions associated with the block. Scheduling of branches only becomes an issue for very advanced compilers that implement superblock scheduling methods.

2.1.2 Latency Considerations

The simple model described above applies equally well to both TriCore 1.2/1.3 and TriCore 2.0 implementations. However it only deals with parallel issue capabilities and does not consider instruction scheduling to avoid pipeline stalls caused by instruction latencies.

Scheduling around instruction latencies is not a dominant consideration for current TriCore code generation. For the most part, instructions execute in one cycle and forwarding in the pipeline enables the result from one instruction to be used by its immediate successor. However there are exceptions to this model, and correctly describing them requires a more detailed view of the pipeline. At this level we encounter some significant differences between the TriCore 1.2/1.3 and TriCore 2.0 pipelines.

2.1.2.1 Floating Point Unit (FPU) Pipeline

When present, the FPU in TriCore 1.3 is implemented using a coprocessor (COP) port on the IP pipeline. Each FPU instruction issued will stall the processor for a number of cycles dependent upon the instruction being executed (see "TriCore 1 32-bit Unified Processor Floating Point Unit (FPU)"). The implication of this is that FPU instructions scheduled for a TriCore 1.3 implementation can be treated identically to single cycle IP instructions.

In TriCore 2.0 the situation is different in that the FPU has been integrated into the core and behaves as if it is part of the IP pipeline. The compiler writer should therefore schedule FPU instructions as multicycle IP instructions, which are described in the following sections.

2.1.2.2 TriCore 1.2/1.3 Regular Integer versus MAC Pipelines

TriCore 1.2/1.3 implementations employ what is predominantly a four-stage pipeline. The multiply-accumulate (MAC) instructions however, employ five pipeline stages. The four stages of the main TriCore 1.2/1.3 integer pipeline are described in **Table 2**, below, while **Table 3** describes the stages used for the 16-bit MAC operations.

Table 2 TriCore 1.2/1.3 Integer Pipeline Stages

Name	Activities
Fetch	Instruction Fetch and predecode
Decode	Instruction decode, operand fetch
Execute	instruction execution
Writeback	ALU result to registers

Table 3 TriCore 1.2/1.3 MAC Pipeline Stages

Name	Activities
Fetch	Instruction fetch and predecode
Decode	Instruction decode, operand fetch
Execute_1	Multiplication
Execute_2	Accumulation
Writeback	Accumulator result to registers

For the multiply-accumulate instructions, the execute stage of the regular pipeline is expanded to two cycles (Execute_1 and Execute_2), and the writeback stage is delayed by one cycle. This means that there will be contention for the result bus between any MAC instruction and a regular integer pipeline (IP) instruction that follows it. The regular IP instruction will stall for one cycle. Therefore a regular IP instruction should not be scheduled in the next instruction cycle following a MAC, if there is any choice.

When a multiply-accumulate (MAC) instruction is followed by another multiply-accumulate, there is no contention for the result bus, and no forced stall. Furthermore, result forwarding from the accumulation stage of one MAC to that of an immediately following MAC allows a dependent MAC to issue immediately, as long as the dependence is on the accumulator input. However, If it uses the accumulator result from the preceding operation as a multiplier input, the second MAC operation will stall for one cycle.

2.1.2.3 TriCore 1.2/1.3 Regular Integer versus MAC Pipelines

In TriCore 2.0 implementations there is no writeback contention between regular IP instructions and MAC operations. The scheduling consideration after a MAC operation described above, does not apply. Both instruction subclasses use two execution stages, with writeback after the second execution stage. Most regular IP instructions do still develop their results in the first execution stage, but the result is staged for one cycle to keep writebacks aligned. Forwarding from the first execution stage is provided, to avoid dependency stalls.

2.1.2.4 TriCore 1.2/1.3 Load-to-Use Latencies

Table 4 describes the stages of the TriCore 1.2/1.3 load-store pipeline.

Table 4 TriCore 1.2/1.3 Load-Store Pipeline Stages

Name	Activities
Fetch	Instruction Fetch and predecode
Decode	Instruction decode, Effective Address (EA) calculation
Execute	AAU (Address Arithmetic Unit) instruction or Load execution
Writeback	AAU result writeback or Store execution

The Effective Address (EA) for a Load or Store operation is computed in the instruction decode cycle. For Loads, the computed EA is presented to the Data Memory Unit (DMU) at the start of the execute cycle. If the requested EA hits in the cache, or resides in the on-chip scratchpad SRAM, the requested data value is returned to the CPU at the end of the execute cycle. Load forwarding in the CPU enables immediate use in the execute stage of the next instruction. *This means that for data register loads, there are no load-to-use delay cycles that the compiler needs to schedule around.*

If the data item referenced is not in the cache or data scratchpad, there will be some number of wait cycles before the data item is returned, but there is nothing that the compiler can do about that. Loads, in all TC1 implementations, are blocking operations. A Load from slow memory, or a cache miss, will block any further instruction issuing until the Load completes, so there is no advantage to scheduling non-dependent operations between a Load and the first use of the data item loaded.[1)]

Note: Unlike data register loads, TriCore 1.2/1.3 address register loads do have a built-in delay cycle between the Load and the use of the loaded value in an address

[1)] On the other hand, there is no disadvantage, and such a scheduling policy could pay off in the future.

calculation. That is because effective addresses are computed in the decode stage, rather than the execute stage. The compiler should always try to schedule a non-dependent operation into that delay slot.[1)]

2.1.2.5 TriCore 1.2/1.3 Definition-to-Store Latencies

For stores, the computed EA (Effective Address) is staged for one or more cycles, before being presented to the DMU (Data Memory Unit) along with the store data. The normal case is one stage of delay, with presentation to the DMU at the end of the execute stage. However if a Load follows a Store, the two instructions contend for DMU access. Priority is given to the Load, even though it follows the Store. The Store data is captured in a write buffer and the Store operation stays pending until the DMU interface is available. Potentially that can be many cycles later, if a Store is followed by a sequence of Loads. However a subsequent Load that references data held in the write buffer is detected and returns the same value as if the Store had been executed at its original point in the instruction sequence.

Because Store data is presented at the end of the execute state, it is possible to forward an ALU (Arithmetic and Logic Unit) result to the Store bus and so enable the parallel issue of a single-cycle ALU instruction and a dependent Store. The same applies for AAU (Address Arithmetic Unit) results in the Address registers. All address arithmetic instructions are single-cycle operations.

For MAC operations, which require a second execution stage, it is not possible to have a dependent store issuing in parallel. A dependent store can issue in the next cycle after the MAC. Further, if an initial MAC operation is followed by a second MAC to a different accumulator, a store of the first accumulator will be able to issue in parallel with the second MAC operation. Hence the sequence *M1-S1-M2* (where *M1* and *M2* are MAC instructions, and *S1* is a store dependent on *M1*) will require three cycles, whereas *M1-M2-S1* will require only two.

[1)] Contrary to what one might expect, there is no corresponding delay slot following an AAU instruction that updates the value of an address register. Loads arrive very late in the execute cycle, whereas AAU results are available early enough to permit forwarding to a dependent EA calculation in the decode stage of the next instruction.

2.1.2.6 TriCore 2.0 Load-to-Use Latencies

Table 5 describes the stages of the TC2 Load-Store pipeline, starting from the decode stage.

Table 5 TriCore 2.0 Load-Store Pipeline Stages

Name	Activities
Decode	Instruction decode and register access
Execute-1	Effective Address (EA) calculation and MMU translation
Execute-2	Memory access
Writeback	Load or AAU (Address Arithmetic Unit) result writeback

To enable higher frequency operation and a physically indexed cache, the effective address for a Load or Store operation is delivered to the memory address bus at the end of the execute-1 stage, rather than the end of decode. Memory access, for cached data and scratchpad, is a "1.5 cycle" operation. The Load data are latched, at the 64-bit memory output interface, at the end of the execute-2 cycle. The first half of the writeback cycle is used for alignment, sign extension and transport of the memory data to the CPU. Load results are therefore potentially available for use in the latter half of the writeback cycle.

This means there is a two-cycle load-to-use latency delay penalty, since all ALU and AAU operations require input operands to be available at the *start* of their first execute stages. *Therefore any instruction requires that there are two delay slots between a register Load and any use of the loaded value.* The compiler should attempt to schedule non-dependent operations into those slots.

2.1.2.7 TriCore 2.0 Definition-to-Use Latencies

In the TriCore 2.0 pipeline, a *Store* of a register result can always issue in parallel, behind the IP or LS instruction defining the result (assuming that the Store is not stalled by an address register dependency). The Store does not go directly to memory but goes into a store buffer, from which its value can be read, if the associated address is subsequently referenced in a *Load* instruction. The definition-to-store latency is therefore zero cycles.

For uses other than the source operand for a Store, non-zero definition-to-use latencies do apply. If the definition is for a data register and the use is in the Load-Store pipeline, then the latency will be two cycles. In the case of a conditional branch on a data register value for example, it will delay the branch resolution by two cycles.

2.1.2.8 Multi-Cycle Instructions

Some TriCore instructions are multi-cycle instructions. This means that they consume multiple instruction issue cycles. They differ in that respect from the 16-bit MAC instructions, which use an added pipeline stage but only one issue cycle. The multi-cycle instructions effectively reissue themselves one or more times. Those that execute in the integer pipeline permit parallel issue of a Load-Store pipeline instruction, but only on the *last* issue cycle of their execution.

The principle multi-cycle instructions the compiler will issue are the 32 x 32-bit multiplies and multiply-accumulates, and the *DVSTEP* instruction. The *CALL* and *RET* instructions are also multi-cycle operations, but they are special cases with their own scheduling considerations. They are discussed further in the next section (**Scheduling Around CALL and RET**). The 32 x 32-bit multiplies and multiply-accumulates require two cycles. *DVSTEP* requires five cycles in TriCore 1.2/1.3, or three cycles in TriCore 2.0.

The fact that a particular instruction is a multi-cycle instruction currently has no direct relevance for instruction scheduling. Nothing will issue in parallel with it, until its last issue cycle. On that cycle it behaves like a regular single-cycle instruction. The extra cycles that it entails add equally to any possible scheduling path.

2.1.2.9 Scheduling Around CALL and RET

A *CALL* is both a branch and a multi-cycle store of the upper context registers. If the call target is resident in the instruction cache, then the first instruction in the called function will be ready to execute one cycle before the context store operation has cleared the pipeline. If the first instruction is an LS (Load/Store) pipeline instruction (e.g. a decrement of the Stack Pointer), it will stall for one cycle. If it is an integer pipeline instruction it will issue immediately, but there will be no available slot for parallel issue of an LS instruction behind it.

A *RET* (Return from Call) combines a branch and a multi-cycle Load of the upper context registers. Although a regular Load will overtake a preceding store with priority access to the DMU, context loads are handled differently. The context Load associated with a *RET* operation will stall until the Store has completed. If there is any choice, the compiler should therefore avoid scheduling a Store operation immediately before a RET[1)].

Following a *RET*, the last cycle of the context load may still be in the pipeline, and the scheduling considerations are similar to those that apply at the start of a called function; i.e. an LS pipeline instruction will stall for one cycle, while an integer pipeline instruction will issue, but with no slot available for parallel issue of an LS pipeline instruction.

1) In the case of functions with no return value, a store as the last operation preceding the return is almost inevitable; almost any other operation that a compiler would normally generate would be rendered dead by the return. The only exceptions would be conditional branches around the return, and system instructions (e.g. *ENABLE*) issued in translation of an intrinsic function call.

2.1.3 Block Ordering and Alignment

The previous section dealt with issues that affect the scheduling of code within basic blocks. It described the pipeline model, as it appears when instruction issue is not limited by the availability of code in the instruction fetch buffers.

For both TriCore 1.2/1.3 and TriCore 2.0, the bus between the instruction fetch unit and the Program Memory Unit (PMU) is 64-bits wide. That is the same as the maximum width of instructions that can be issued in one cycle, so in straight-line code execution out of the scratchpad code memory, availability of code will never limit instruction issue. But when branches are present, or when execution is from slower memory or cached memory with miss possibilities, that ceases to be true.

Considerations regarding the operation of the instruction fetch unit in the presence of branches and cache misses do have some impact on the compiler. They can affect the optimal ordering of basic blocks within a function. For TriCore 1.2/1.3, they can also affect the insertion of no-ops or the deliberate use of 32-bit instruction forms to force the alignment of loop start locations.

2.1.3.1 TriCore 1.2/1.3 & 2.0 Instruction Fetch Unit

All implementations perform best when fetch targets are aligned on 64-bit boundaries. This enables the instruction fetch to guarantee at least two instruction issues and hence a possible dual issue. Aligning branch or call targets on 32-bit may only result in a single issue even if the following instruction can be effectively dual issued.

In TC1.3 implementations, the PMU interface supports doubleword fetches from halfword boundaries. If the fetch crosses a cache line boundary, a second cycle will still be needed to return the portion of the doubleword residing in the second cache line. However, with 256-bit cache lines, only about one in five randomly aligned instruction fetches will incur the penalty. This greatly reduces the average performance loss due to randomly aligned branch targets (Prefetching into a streaming buffer normally hides line-crossing penalties for straight-line code).

Unfortunately this feature does not dispose of the worst-case scenario of a loop whose start address happens to fall at the last halfword within a cache line. In order to guarantee performance of the most critical loops, the compiler must still provide users with some means to force a loop to be aligned to a doubleword boundary.

2.1.3.2 TriCore 1.2/1.3 Branch Timings

In TriCore 1.2/1.3, the branch target address is always computed in the instruction decode cycle. If the branch is unconditional, or predicted as taken, the target address is sent to the instruction fetch unit at the end of the decode cycle. If the target address is resident in the instruction cache or in the code SRAM, the instruction words at that address are returned from the PMU to the instruction fetch unit near the end of the next cycle, which is the execute cycle for the branch instruction. In parallel, if the branch was conditional, the actual branch direction is resolved. If the branch direction is contrary to the prediction, then the correct target address is sent to the fetch unit at the end of the execute cycle.

The decode cycle for the first instruction at the branch target address follows the execute cycle for the branch instruction. This is equivalent to a branch time of two cycles for a branch that is either unconditional, or correctly predicted as taken. If a branch is incorrectly predicted, the correct target address is sent to the instruction fetch unit at the end of the branch instruction's execute cycle, giving an effective branch time of three cycles. For a branch that is correctly predicted as not taken, the execution time is the same as for a regular instruction: one cycle.

The branch timing numbers are summarized in **Table 6**, which follows.

Table 6 TriCore 1.2/1.3 Branch Timings

Predicted Direction	**Actual Direction**	
	Taken	**Not Taken**
Taken	2	3
Not Taken	3	1

2.1.3.3 TriCore 2.0 Branch Timings

The TriCore 2.0 branch mechanism is compliant with TriCore 1.2/1.3, but is implemented differently. Branch instruction selection and hence block ordering should be performed in the same manner as for TriCore 1.2/1.3 to limit the differences in compiler implementations.

2.1.3.4 Block Ordering for TriCore 1.2/1.3

Compilers often have some knowledge of branch probabilities. That knowledge may be derived from heuristics applied to the source code, or from direct feedback of profiling information. In either case the compiler can use the information to reduce branch costs. The conditional branch instructions of the TriCore architecture do not include a static prediction bit as such, but the compiler can choose to order code blocks in such a way that the static prediction scheme employed by the implementation becomes "self-full filling". That scheme is:

- Backward and short forward branches taken
- Long forward branches not taken

A short forward branch is a 16-bit branch instruction with a positive displacement.

Although it is common to order blocks for code generation according to their natural order in the source code, there is no inherent requirement for the compiler to do so. Ordering according to the source code does facilitate debugging, and it normally gives rise to the minimum number of unconditional branches required for control flow connectivity. However, given information on individual branch probabilities, performance can usually be improved through informed ordering of the blocks.

Determination of an optimal block ordering is in principle, an NP-complete[1] problem. However it yields reasonably well to simple heuristic approaches. An exploration of algorithms for block ordering is beyond the scope of this document, but the most important heuristic rule is simple and intuitive:

- When a control flow path forks, try to make the more probable path the fall-through case. In other words, select, as the next block in the code output sequence, the successor that is most likely to be executed.

Under this strategy, the conditional branch to the less frequently executed successor will normally become a long forward branch, implicitly predicted as not taken. The cost of the branch instruction will be only one cycle, when the prediction of *not taken* is correct. The code that is executed will also tend be more localized, resulting in more efficient use of the instruction cache.

Moving infrequently executed blocks to the end of a function, out of the natural order in which they appear in the source code, does have a small cost in code space. The out-of-line block will typically require an unconditional branch at its end, to rejoin the main flow path, where it might otherwise be able to fall through to the main path. Block reordering according to execution frequency is therefore a space-time trade-off, subject to compilation flags specified by users.

The code space cost for optimized block ordering can be reduced by exploiting the availability of *short forward branches*. These are 16-bit branch forms that are implicitly predicted as *taken.* A block ordering strategy, adjusted to take advantage of short forward branches, can be summarized as follows:

- Given a control flow subgraph of the form illustrated in **Figure 1** (representing an *if* statement), and with the code size associated with block B less than eight words, always select block B as the physical successor to block A in the code output sequence.
- If the path from A to B is taken infrequently compared to that from A to C, terminate block A with a short conditional branch around B (statically predicted as taken).
- If the path from A to B is taken more frequently than that from A to C, and the optimization mode is for time, use a long form conditional branch at the end of A (statically predicted not taken), even though a short branch form would reach.

[1] NP is the class of decision problems that a Nondeterministic Turing machine can solve in Polynomial time. NP-complete (Nondeterministic polynomial time complete) is a subset of NP, with the additional property that if solved in polynomial time it would be possible to solve all problems in that NP class.

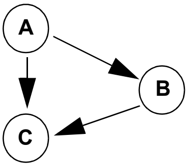

Figure 1 Control Flow Subgraph for *if* Statement

2.2 Pipeline Balancing

When the initial translation for a block contains unused integer or LS (Load/Store) pipeline issue slots (i.e. the block contains predominantly one type of instruction or the other, rather than a balanced mix that can be paired for parallel issue), there are various transformations that the compiler can apply to improve the balance and reduce overall execution time. These transformations are implementation dependent in that they only make sense for implementations similar to the current ones, where parallel issue of instructions to the integer and LS pipelines is supported. These "pipeline balancing" transformations include the following (described below):

- **Data Literals**
- **Speculative Load/Store (LS) Operations**
- **Speculative Integer Operations**

Data Literals

When a register copy of a data literal value is required, the compiler has a choice of implementation methods. It can load the value from the literal pool or, depending on the width of the literal, it can use one or two immediate value Load instructions. If the block is dominated by integer pipeline instructions, loading the value from the literal pool may be preferable, as the Load can issue behind one of the integer pipeline instructions. If the block is dominated by LS pipeline instructions, immediate value loads may be preferable, even if the literal is a 32-bit value that requires two instructions to load. The immediate value load instructions are integer pipeline instructions and can issue in the otherwise unused issue slots preceding LS pipeline instructions.

Speculative Load/Store (LS) Operations

When a block has an excess of LS pipeline instructions, and its only control flow predecessor has an excess of integer pipeline instructions, execution time may be reduced, at no cost in code space, by promoting one or more of the LS pipeline instructions to the predecessor block, where they can issue in parallel behind one of that block's excess integer pipeline instructions. The promoted instruction(s) must obviously have no dependencies on operations within the block from which they are moved, and they cannot be Loads or Stores involving variables declared *volatile*.

Here it is assumed that the predecessor block has multiple successors (commonly two), and that the result of the promoted operation is not used in every successor. Otherwise the promoted operation is not speculative at all, and its promotion amounts to simple code hoisting. Code hoisting is a standard code-space optimization that should be supported by any competent compiler. It reduces code space with no increase in execution time, by replacing multiple copies of an operation with a single copy. Speculative promotion on the other hand, leaves the code size unchanged, but moves an operation to a point where it can issue in a slot that would otherwise go unused. The promoted operation may be executed at times when it needn't be, but because it issues in a slot that would otherwise be idle, there is no execution time penalty.

If the promoted LS pipeline operation is an *LEA* instruction or an AAU instruction (such as *ADD.A*), then there are very few complications to consider. The result is produced in an Address register and there are no side effects. The only potential problem is an increase in register pressure, due to the increased lifetime of the Address register holding the result of the operation. If the promoted operation is a Load however, the compiler must guarantee that speculative execution of the Load cannot cause an invalid address trap. The possibility of a cache miss must also be considered. In some cases, the compiler may be able to rule out a cache miss, because it knows that the access is to uncached scratchpad memory, or because the Load is preceeded by other accesses that will map to the same cache line. In other cases the compiler will not be able to rule out a cache miss, but profiling information or static analysis may indicate that the block from which the operation was promoted is the overwhelmingly predominant successor of its predecessor. In that instance promotion can pay off, even if it does lead to a cache miss.

Speculative Integer Operations

Integer pipeline operations can also be speculatively promoted for better pipeline balance and improved execution speed. The appropriate conditions are a block with an excess of integer pipeline instructions, whose only control flow predecessor has an excess of LS (Load/Store) pipeline operations. In that case, execution cycles can be saved by promoting one or more of the integer pipeline operations from the successor block to the predecessor.

3 Advanced Optimizations

3.1 SIMD Optimization - An Introduction

TriCore exploits the data parallelism present in a large number of DSP algorithms by executing the same operation on different data elements in parallel. This is accomplished by packing several data elements into a single register and introducing a new type of instruction to operate on packed data. All packed data types are 32-bits wide. Two new data types are supported:

- Packed byte
- Packed halfword

(*For additional information, please refer to the TriCore Architecture manual*)

Packed arithmetic instructions can, potentially, improve the speed of the TriCore application from two to four times.

An effective TriCore compiler must expose data parallelism available in DSP algorithms and generate packed arithmetic assembly code. Data parallelism is typically found in loops. ANSI C does not provide any parallel constructs in the language. This requires the TriCore compiler to automatically extract the data parallelism from the *for* and *while* loops. During the parallelism extraction, original semantics of the application should not be violated. *Data and control dependence analyses* guide the series of parallelizing transformations to keep the original semantics of the application. These are the same techniques compilers for vector and SIMD machines have used. The use of packed arithmetic in TriCore is very similar to the exploitation of parallelism in SIMD machines[1)], such as Thinking Machines Corporation's CM-1.

Before discussing various transformations, it is important to note that the packed arithmetic exploitation is not a single compiler step but a series of steps. The goal for packed arithmetic exploitation is to improve the performance of the application. Some of these intermediate steps are expensive and if the succeeding steps successfully generate the packed arithmetic code, the overall performance is improved. However in some cases, if the succeeding steps do not generate the packed arithmetic code, the performance of the program may be degraded. So a TriCore compiler must ensure that either early analysis is performed to completely avoid the expensive intermediate steps on the loops that are not parallelizable, or apply the appropriate inverse transformations on the non-parallelizable loops. The order of applying various transformations is also important, and will significantly determine the amount of parallelism extracted. This *phase ordering* of transformations should be carefully selected to extract as much parallelism as possible.

1) TriCore does not support masking instructions and conditional operations typically supported in SIMD machines like CM-1.

3.2 Data Dependence Analysis

C programs impose certain dependencies on variables (scalar and array) to be written and read in a particular order. As long as the order of the reads and writes on these variables is preserved, the rest of the operations can be executed in any order, or even concurrently. *Dependence analysis,* in essence, abstracts this order of execution or dependences on the variables. Consequently the smaller the number of dependences in the original application, the greater the opportunity for available parallelism.

There are three types of data dependences:

- Flow dependence occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement.
- Anti-dependence occurs when a variable is used in one statement and reassigned in a subsequently executed statement.
- Output dependence occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement.

Anti-dependence and output dependence arise from the reuse or reassignment of variables, and are called *false dependences*. The false dependences can be rearranged creating opportunities for parallelism extraction. Rearrangement of false dependencies is one of the concepts many parallelizing transformations exploit. Another technique is to rearrange different loops, as long as no original dependences are violated, in a nested loop to exploit the parallelism.

Detecting the data dependences in loops requires array subscript analysis, also called *dependence testing*. Choosing the appropriate dependence testing techniques will determine the complexity and the speed of parallelism extraction. The intention in this chapter is to describe and illustrate the useful application of SIMD transformations for generating the packed arithmetic code, but not to detail how to perform dependency analysis to implement them. However the relevant dependence analysis information is provided as is necessary for the discussion of the restructuring transformations.

For a thorough treatment on dependence analysis and automatic parallelization, please refer to the books referenced in the **Bibliography** at the end of this chapter.

Note: Throughout this chapter it is assumed that the data element sizes of the arrays in the example codes are either byte or half-word.

3.3 FORALL Loops

FORALL is a loop construct in the Fortran 90 standard. The **for and FORALL Loop Order of Execution** table below, illustrates the differences in the order of execution of C *for* loop and Fortran 90 *FORALL* Loops.

- In the C *for* loop, the semantics stipulate the complete body of the loop to be executed for the each loop index before the succeeding loop index.
- The *FORALL* loop semantics require each statement of the loop body in the lexical order to be executed for all the loop indices before the succeeding statement.

Table 7 *for* and *FORALL* Loop Order of Execution

***for* Loop**	**FORALL Loop**

`for (I = 0; I < 100; I++) {`
`      ST1;`
`      ST2;`
`      ST3;`
`}`
	`FORALL (I = 0; I < 100; I++) {`
	`        ST1;`
	`        ST2;`
	`        ST3;`
	`}`

for Loop Order of execution
 	
`I = 0 ->   ST1, ST2, ST3`
`I = 1 ->   ST1, ST2, ST3`
`..................................`
`..................................`
`I = 99 -> ST1, ST2, ST3`
	FORALL Loop Order of execution
	
	`ST1 -> I = 0, 1, 2, .., 99`
	`ST2 -> I = 0, 1, 2, .., 99`
	`ST3 -> I = 0, 1, 2, .., 99`

To generate packed arithmetic code for TriCore, C loops should be transformed into equivalent *FORALL* loops without violating the original semantics. Theoretically it may not be possible to convert a given *for* loop into a semantically equivalent *FORALL* loop, and vice-versa. Dependence analysis determines the legal convertibility of a C loop into a *FORALL* loop. A *for* loop can be transformed into a semantically equivalent *FORALL* loop if there are no dependence cycles and no self loop carried flow dependences in the inner most loop.

At this point two important observations should be noted:

1. Since the maximum degree of parallelism exploitable in TriCore is four, it is not necessary to convert the complete *for* loop to a *FORALL* loop. Where a strip-mined loop with strip size of four can be converted to a *FORALL* loop, it is sufficient to generate packed arithmetic for the TriCore. Strip-mining is explained in detail in the next section (**Section 3.4 Strip-mining**).
2. The second observation is that even if the innermost loop can be converted to a *FORALL* loop, if all the array footprints are not allocated in contiguous memory then packed arithmetic code cannot be generated.

3.4 Strip-mining

Strip-mining is the central transformation step in the packed arithmetic code generation. To give a flavor of exposing packed data operations, consider the loop in the following figure:

```
for (I = 0; I < N; I++) {
     A[I] = B[I] + C[I];
}
```

Figure 2 Example Loop to Show Strip-mining

The performance of this loop can be improved between two or four times depending on whether the data element sizes are either half-word or byte, respectively. For this loop the TriCore compiler should be capable of performing strip-mining and then generate packed data instructions.

After Strip-mining this loop with strip of size *S* (*S* = 2 or 4 in TriCore depending on the array element being a byte or half-word respectively), it becomes:

```
X = N%S;
for (I = 0; I < X; I++) {
   A[I] = B[I] + C[I];
}
for (IS = X; IS < N; IS=IS+S) {
   for (I = IS; I < IS+S; I++) {
       A[I] = B[I] + C[I];
   }
}

Note: N%S ≡ N&(S-1) when S is a power of 2
```

Figure 3 Loop After Strip-mining the Loop of Figure 2

The first loop is to handle the cases where the number of iterations is not a perfect multiple of *S*. This loop has to be executed in the regular non-packed arithmetic.

If *N* is compile-time known constant and is a perfect multiple of *S*, then there is no need to generate the first loop. In this case there will be no additional overhead and the loop's performance will be improved to the maximum.

In the doubly nested loop, the inner loop is the candidate for packed arithmetic code assuming that this inner loop can be converted to a FORALL loop and that the footprint of each array is in contiguous memory (as explained in the previous section). If these two conditions are satisfied, the inner FORALL loop will be transformed into the Triplet notation as shown in **Figure 4**. Finally the pseudo-assembly version of the loop will be generated as shown in **Figure 5**.

Note: A comprehensive list of packed arithmetic instructions corresponding to regular DSP instructions can be found in Appendix A of this document.

```
for (IS = X; IS < N; IS=IS+S) {
   A[IS:IS+S-1:1] = B[IS:IS+S-1:1] + C[IS:IS+S-1:1];
}
```

Figure 4 Triplet Notation Code for Parallel Section in the Loop of Figure 3

```
for (IS = X; IS < N; IS=IS+S) {
   LD.W D0, B[I];
   LD.W D1, C[I];
   ADD.[BH] D2, D0, D1;
   ST.W A[I], D2
}
```

Figure 5 Pseudo TriCore Packed Arithmetic Code for the Loop Example

Strip-mining is always legal in C loops if the destination does not overlap with the source(s). Since it is transforming a single loop into two, it creates more loop overhead. So if packed arithmetic cannot be generated for the inner loop, it is wise not to perform strip-mining in the first place.

3.5 Scalar Expansion & Loop Peeling

When scalars are assigned and used in loops they create spurious dependence cycles. In most of these cases a scalar assigned in each iteration will be read in the same iteration. So *expanding the scalar* into array will remove these dependence cycles. Consider the loop in below:

```
for (I = 0; I < N; I++) {
     T = B[I] + C[I];
     A[I] = T + T*D[I];
}
```

Figure 6 Example Loop to show Scalar Expansion

This loop cannot be parallelized because of the scalar variable *T*.

Applying the scalar expansion transformation, the loop will be transformed as shown in Figure 7.

```
for (I = 0; I < N; I++) {
     TP[I] = B[I] + C[I];
     A[I] = TP[I] + TP[I]*D[I];
}
T = TP[N-1];
```

Figure 7 Loop After Applying Scalar Expansion to the Loop of Figure 6

Now the loop can be strip-mined and packed arithmetic code can be generated.

In some cases the scalar will be used first before being assigned in the loop, where the application of *loop peeling* will adjust the iterations so that scalar is assigned and used respectively.

Consider the following example: There is a loop-carried anti-dependence due to the scalar *T* and the loop cannot be converted to a *FORALL* loop. An example loop in shown in **Figure 8** and the transformed loop after *loop peeling* transformation is shown in **Figure 9**.

```
for (I = 0; I < N; I++) {
     A[I] = T + C[I];
     T = C[I] * (D[I] + B[I]);
}
```

Figure 8 Example Loop to Show Loop Peeling

```
A[0] = T + C[0];
for (I = 1; I < N; I++) {
    T = C[I-1] * (D[I-1] + B[I-1]);
    A[I] = T + C[I];
}
T = C[N-1] * (D[N-1] + B[N-1]);
A[N-1] = T + C[N-1];
```

Figure 9 Loop After Applying Loop Peeling to Loop of Figure 8

The application of scalar expansion will remove the dependence cycles and the loop can then be converted to a *FORALL* loop.

In the third case, if a scalar is read in a loop, the scalar has to be expanded to generate packed arithmetic code.

Consider the loop in the table below. To convert this loop to a *FORALL* loop, scalar *T* has to be expanded as shown in the previous example. Since *TP[I]* is loop independent, code motion should be applied. The emulation of loading scalar *T* from memory to a register and expanding the scalar in TriCore register is also shown below. In this loop, code motion saves these three instructions in each iteration. When compared to the non-*FORALL* loop this gives a net saving of one load instruction per iteration.

`for (I = 0; I < N; I++) {`
`     A[I] = T + C[I];`
`}`
1. Example Loop
	`for (I = 0; I < N; I++) {`
	`     A[I] = TP[I] + C[I];`
	`}`
	2. Scalar Expansion

`ld     T, D0`
`insert D0, D0, D0, 8, 8`
`insert D0, D0, D0, 16, 16`
3. Emulation of Scalar expansion	

3.6 Loop Interchange

With multi-nested loops there is an opportunity to convert any of the loops to a *FORALL* loop. However the loop to be converted should be chosen on the basis that the array memory accesses are contiguous. If the corresponding loop is not the innermost loop, it may need to be interchanged with the current inner-most loop, provided that it does not violate any of the original dependences. Loop interchange has potentially large performance benefits.

Consider the doubly nested loop given in the following figure:

```
for (J = 0; J < N; J++) {
     for (I = 0; I < N; I++) {
      A[I][J] = B[I][J] + C[I][J];
     }
}
```

Figure 10 Example Loop to Show Loop Interchange

Since the arrays in C are stored in row-major order, strip-mining the inner loop (the *I* loop) is not sensible because it does not improve the performance in anyway. However the loops can be interchanged as shown in **Figure 11**:

```
for (I = 0; I < N; I++) {
     for (J = 0; J < N; J++) {
      A[I][J] = B[I][J] + C[I][J];
     }
}
```

Figure 11 Loop After Applying Loop Interchange to the Loop of Figure 10

The *J* loop can be strip-mined and converted to a *FORALL* loop for packed data performance.

3.7 Loop Reversal

Sometimes a loop cannot be converted to a *FORALL* loop if the loop index is decreasing; i.e. the array elements are accessed in the reverse order. This does not correspond well with the packed arithmetic array accesses.

Consider the example loop in **Figure 12** and the transformed loop after applying loop peeling in **Figure 13**:

```
for (I = N-1; I > -1 ; I--) {
    A[I] = B[I] + C[I];
}
```

Figure 12 Example Loop to Show Loop Reversal

```
for (I = 0; I < N ; I++) {
    A[I] = B[I] + C[I];
}
```

Figure 13 Loop After Applying Loop Reversal to Loop of Figure 12

In multi-nested loops, loop reversal can sometimes make the loop interchanging legal for converting to a *FORALL* loop. This is a very useful application for performance improvement.

3.8 Loop Fusion and Loop Fission

When two adjacent loops have the same loop limits, they can be *fused* into one loop. This offers many advantages to improve the performance of the loops. Loop control code is halved and the loop can therefore run up to twice as fast. Temporal locality of the code will be improved as long as the combined loop does not exceed the size of the instruction cache. There can also be other opportunities for common sub-expression elimination and instruction scheduling. However loop fusion is not always legal. The fused loop should not create any backward dependences.

Consider the two adjacent loops in **Figure 14** and Loop Fusion applied to it as shown in **Figure 15**:

```
for (I = 0; I < N; I++) {
     A[I] = B[I] + C[I];
}
for (I = 0; I < N; I++) {
     D[I] = A[I] + C[I];
}
```

Figure 14 Example Loop for Loop Fusion

```
for (I = 0; I < N; I++) {
     A[I] = B[I] + C[I];
     D[I] = A[I] + C[I];
}
```

Figure 15 Loop After Applying Loop Fusion to the Loop of Figure 14

The loading of arrays *A* and *C* for the second statement are completely eliminated because they are already available in the registers. Obviously the loop can be strip-mined and converted to a *FORALL* loop for more performance.

Loop Fission is the inverse of Loop Fusion. If the loop body is big, applying Loop Fission sometimes helps increase the cache locality when the cache is small, and also avoids the register spilling.

Both Loop Fusion and Loop Fission are legal only if the dependences permit them.

3.9 Reductions

Reduction loops are those where a vector is reduced into a scalar. One example is a sum or product of the elements of a row in a matrix or a vector. Another example is vector dot product. TriCore supports 16-bit multiplications and multiply accumulates but not 8-bit versions in packed arithmetic. However in the case of multiply accumulates TriCore provides a very powerful feature where the multiply accumulate instruction adds the two accumulates in the same cycle for one single result. This eliminates the need for the epilogue code to accumulate the partial accumulates. There are many important kernels which are reductions. So it is important that the TriCore compiler has the capability to detect the reduction loops and generate the packed arithmetic.

Consider the following loop:

```
S = 0;
for (I = 0; I < 100; I++) {
      S = S + A[I];
}
```

Figure 16 First Example Reduction Loop

Using dependence analysis the TriCore compiler should recognize that this is a reduction loop.

As in the rest of the chapter, if the data size of *S* and elements of *A* are a byte, the loop can be strip-mined and code can be generated in triplet notation as shown in **Figure 17**.

```
long ST;
ST = 0;
for (I = 0; I < 100; I=I+4) {
      ST = ST + A[I:I+3:1];
}
extr Da, ST, #16, #16
add ST, Da
extr Da, ST, #8, #8
add S, ST, Da
```

Figure 17 Strip-mined Loop with Triplet Code for Loop of Figure 16

In **Figure 17**, *ST* is the temporary long word generated to perform packed arithmetic. *ST* is of the size long word to facilitate packed arithmetic. Since *ST* is long word it can be initialized to zero in one instruction, otherwise it would have loop to initialize *ST*. The f*or* loop in **Figure 17** is the actual reduction performed in packed arithmetic.

At the end of the loop *ST* contains 4 partial reductions (sums in this example) which can be summed up to get the final sum by using the two extracts and two adds. The correct function of this approach requires that either the destination (*S*) is of byte type, or that none of the byte accumulations overflow. Improved performance and flexibility can be achieved using halfword multiply-accumulation (*MADDM.H*) with a unit multiplier.

Now consider the loop in **Figure 18**. This is an example of vector dot product. It is assumed that the data of *S*, the elements of *A* and *B,* are signed 16-bit numbers. The packed arithmetic reduction code can be generated for this loop as shown.

As mentioned previously, the results of the two multiply accumulates in the packed instruction will be summed up in each cycle, eliminating the need for the sum of the two partial sums. The *EXTR* (Extract Bit Field) instruction will put the result in *S* as well as extracting the 16-bit left shifted result from *ST*.

```
S = 0;
for (I = 0; I < 100; I++) {
      S = S + A[I]*B[I];
}
```

Figure 18 Second Example Reduction Loop

```
long long ST;
ST = 0;
for (I = 0; I < 100; I=I+2) {
      ST = ST + A[I:I+1:1]*B[I];
}
extr S, ST, #16, #16
```

Figure 19 Strip-mined Loop with Triplet Code for Loop of Figure 18

3.10 Miscellaneous Transformations

In some cases, even if there is a dependence cycle, loop can be strip-mined and converted to a *FORALL* loop if the dependence distances are more than 3 for byte packed instructions and 1 for half word packed instructions.

```
for (I = 8; I < N; I++) {
        A[I] = B[I-8] + C[I+1];
        B[I] = A[I-5] + D[I];
}
```

Figure 20 Example Loop with a Dependence Cycle

```
for (IS = 8; IS < N; IS=IS+4) {
    FORALL (I = IS; I < IS+4; I++) {
        A[I] = B[I-8] + C[I+1];
        B[I] = A[I-5] + D[I];
    }
}
```

Figure 21 Loop after Strip-mining and Converting to FORALL

In some rare applications, Index-set splitting is another transformation which can convert some loops to *FORALL* loops. This is different from Loop Fission. Loop Fission cuts the loop body into two parts using the same iterations on both the parts. Index-splitting splits the iteration space into two parts but executes the complete body in both the loops.

In general, a series of transformations such as those described above may need to be applied to achieve maximum performance for a nested loop, and a TriCore compiler must use packed data instructions to exploit maximum performance wherever possible.

Bibliography

1. Zima, Hans, and Chapman, Barbara. [1990] Supercompilers for Parallel and Vector Computers, ACM Press Frontier Series, New York, NY, ISBN 0-201-17560-6.
2. Wolfe, Michael. [1996] High Performance Compilers for Parallel Computing, Addison-wesley, Redwood City CA, ISBN 0-8053-2730-4.

4 DSP Support

DSP support for TriCore has been approached from two different sides: one is utilization of TriCore DSP Instruction set and the second is DSP programmability in a high-level language like ANSI C. ANSI C/C++ language is not designed for DSP programming and does not have any SIMD constructs.

The essential aim is to program TriCore from a high-level Language like C/C++ so that the full power of TriCore is utilized in the DSP applications, through effective use of the TriCore DSP and packed arithmetic instructions. Another aim is to protect the DSP software investment in TriCore; i.e. the portability of the code to the next generation TriCore processors, and perhaps other processors as well.

This chapter illustrates these issues in detail and the expected solutions from the Compiler vendors.

4.1 The Requirement of DSP-C Language

The most general and long term solution is to extend the C language with DSP language support. ISO C is already working towards this. This chapter is not designed to duplicate the work of the ISO effort, and TriCore by default, chooses to have the ISO DSP-C conformance from the compiler vendors. Specifically the DSP-C extensions being developed to the ISO/IEC IS 9899:1990 standard. The DSP-C extensions for TriCore are briefly described, including the new data types, new qualifiers, the associated keywords, promotion and conversion rules, the syntax and semantic rules and the library support.

4.1.1 Data Types

DSP programming requires fixed point arithmetic. It leads to the definition of new data types __*fixed* and __*accum.* The size (the number of bits) of each of these data types, the limits (the minimum and maximum allowed values) and the EPSILON values (the difference of value between 0.0 and the least value greater than 0.0 that is representable), are described in the proposed ISO DSP-C extensions.

The various forms of these two data types are as follows:

signed short __*fixed*	signed short __*accum*
unsigned short __*fixed*	unsigned short __*accum*
signed __*fixed*	signed __*accum*
unsigned __*fixed*	unsigned __*accum*
signed long __*fixed*	signed long __*accum*
unsigned long __*fixed*	unsigned long __*accum*

Data types can also be extended by the addition of memory-qualifiers. General memory qualifiers are defined as *__X* and *__Y*. However TriCore does not distinguish between *__X* and *__Y* memories and they should therefore be mapped to the unified memory. The use of *__X* and *__Y* memory qualifiers make the code portable to the next generation processors.

An extra qualifier, the *__circ* qualifier, can be added to the pointer types, thereby annotating the pointer to point to a circular array with special address arithmetic behaviour, as explained in the ISO standard.

Examples

```
"int *" denotes "pointer to int".
"int * __X" denotes "pointer to int which pointer value is located in __X
memory".
"__X int *" denotes "pointer to int whose value is located in __X memory".
"__circ __X int * __Y p" denotes "p is a int pointer whose pointer value
is located in __Y memory and p points to a __circ array which is allocated
in __X memory".
```

All fixed point constants are of non-saturated type. To change the saturation-type an explicit type cast should be used.

4.1.2 Keywords

The following are the new keywords introduced:

- __fixed
- __accum
- __circ
- __sat

4.1.3 Other Topics

Without duplicating the effort, it is expected that all the following topics shall be in conformance to the ISO/IEC IS 9899:1990 standard:

- Arithmetic operands
- Fixed point and integral conversion rules
- Fixed point and floating conversion rules
- Conversion and Promotion Rules
- Saturation Promotions
- Pointers
- Array Subscripting
- Function calls
- Structure and union members
- The sizeof operator
- Cast operators
- Multiplicative operators
- Additive operators
- etc.

Appendix A: Instruction Pairs for Packed Arithmetic

This appendix serves as a guide for an assembly programmer or a compiler writer, on the use and capability of TriCore's packed arithmetic instructions. Given an instruction, the following table guides a compiler writer or assembly programmer to choose the corresponding packed arithmetic instruction. The list is comprehensive in the sense that all the complete packed arithmetic instructions are listed.

	Instruction type	**Assembly Mnemonic**	**Corresponding Packed Arithmetic Mnemonic**
1	Absolute value	ABS	ABS.B ABS.H
2	Absolute value of difference	ABSDIFF	ABSDIFF.B ABSDIFF.H
3	Absolute value of difference with saturation	ABSDIFFS	ABSDIFFS.H
4	Absolute value with saturation	ABSS	ABSS.H
5	Add	ADD	ADD.B ADD.H
6	Add signed with saturation	ADDS	ADDS.H
7	Add unsigned with saturation	ADDS.U	ADDS.HU
8	Equal (Mask)	EQ.W	EQ.B EQ.H
9	Equal Any (Flag)	EQ	EQANY.B EQANY.H
10	Load	LD.Q LD.B LD.H	LD.W LD.D
11	Load unsigned	LD.BU	LD.HU
12	Less than	LT	LT.B LT.H LT.W
13	Less than unsigned	LT.U	LT.BU LT.HU LT.WU
14	Multiply Add	MADD(S) MADD(S).Q	MADD(S).H

	Instruction type	Assembly Mnemonic	Corresponding Packed Arithmetic Mnemonic
15	Multiply add -- multi-precision	MADD(S).Q	MADDM(S).H
16	Multiply add with rounding	MADDR(S).Q	MADDR(S).H
17	Maximum value	MAX	MAX.B MAX.H
18	Maximum value unsigned	MAX.U	MAX.BU MAX.HU
19	Minimum value	MIN	MIN.B MIN.H
20	Minimum value unsigned	MIN.U	MIN.BU MIN.HU
21	Multiply sub	MSUB(S) MSUB(S).Q	MSUB(S).H
22	Multiply sub/add -- multi-precision	MSUB(S).Q	MSUBADM(S).H
23	Multiply sub -- rounding	MSUBR(S).Q	MSUBR(S).H
24	Multiply	MUL(S) MUL.Q	MUL.H
25	Multiply -- rounding	MULR.Q	MULR.H
26	Store	ST.B ST.H	ST.H ST.W ST.D
27	Subtract	SUB	SUB.B SUB.H
28	Subtract signed -- saturation	SUBS	SUBS.H
29	Subtract unsigned -- saturation	SUBS.U	SUBS.HU

Appendix B: Coding Examples

This Appendix demonstrates how compactly Tricore assembly code can be written for DSP kernels in ISO DSP-C. The following examples show how TriCore uses its compact and efficient instruction set to program the DSP C kernels.

```
#define N 64
void VectorMult(short __fixed X[N], short
                __fixed Y[N], short __fixed Z[N])
{
 int I;
 for (I = 0; I < N; I++) {
   Z[I] = __round(X[I]*Y[I]);
}
```

Figure A.1: Vector Multiplication

```
lea  a3, (N/4-1)
lea  a2, x2values
lea  a1, y2values
lea  a4, z2values
ld.w d5, [a2+]4
ld.d e6, [a1+]8
vect2loop:  mulr.h  d0,d5,d6ul,#1
ld.d  e4,[a2+]8
mulr.h  d1,d4,d7ul,#1
ld.d  e6,[a1+]8
st.d  [a4+]8,e0
loop  a3,vect2loop
```

Figure A.2: Tricore assembly code of the kernel in Figure A.1

```
#define N 64
void VectorPreempt(short __fixed sX[N], short
                __fixed sV[N], short __fixed sZ[N])
{
 int I; short __fixed sK;
 for (I = 0; I < N; I++) {
   sZ[I] = __sat(__round(sV[I]+sX[I]*sK);
}
```

Figure A.3: Vector Preemphasis

```
lea  LC, (N/4-1)
mov.u  d6,#0x91ec
ld.w  d3,[Xptr+]4
addih  d6,d6,#0x91ec
ld.d  e4, [Vptr+]8

preloop:    maddrs.h  d0,d4,d3,d6ul,#1
ld.d  e2,[Xptr+]8
maddrs.h  d1,d5,d2,d6ul,#1
ld.d  e4,[Vptr+]8
st.d  [Zptr+]8,e0
loop  LC,preloop
```

Figure A.4: Tricore assembly code of the kernel in Figure A.3

h t t p : / / w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG