Five Methods of Utilizing the MIPS® DSP ASE

This paper describes five methods of utilizing the MIPS DSP ASE (Application-specific Extension) to enhance the performance of signal and media processing programs for DSP software applications . These methods include hand-coding in assembly language, using asm macros, intrinsics, and fixed-point data types and operators in C, and compiler auto-vectorization. Programmers can choose methods based on performance goals and ease of programming.

Document Number: MD00783
Revision 01.01
June 5, 2011

Contents

1 Introduction

The MIPS® DSP Application-Specific Extension (ASE) provides enhanced performance capabilities for a wide range of signal-processing applications, with computational support for fractional data types, SIMD, saturation, and other operations that are commonly used in these applications.

This paper describes five methods of utilizing the MIPS DSP ASE, including hand-coding in assembly language, using asm macros, intrinsics, and fixed-point data types and operators in C, and auto-vectorization performed by the compiler. All of these methods are supported by the GNU Compiler Collection (GCC) [15] and GNU Assembler (GAS) [16] for MIPS cores.

2 Five Methods of Utilizing the MIPS® DSP ASE

The GNU Compiler Collection (GCC) and GNU Assembler (GAS) provide the options `-mdsp` and `-mdspr2` that enable the availability of Rev1or Rev2 MIPS DSP ASE instructions respectively for code generation. Use of these options is assumed in the examples in the remainder of this document.

2.1 Assembly

Hand-coding in assembly language is the most time-consuming method of utilizing the MIPS DSP ASE, but can produce code with the highest performance. Programmers can use techniques such as scheduling DSP instructions to avoid pipeline stalls, and using the 32 registers and four accumulators to avoid memory accesses.

To obtain a correct assembler file format, programmers can write a simple C function, then compile it to an assembly file (e.g., `mips-sde-elf-gcc -S func.c`,), and then simply reuse the output.

Assembler functions that are called from C functions must obey the MIPS O32 ABI [14] for 32-bit MIPS cores, including the correct handling of caller-saved and callee-saved registers. However, if assembly functions are only called from other assembly functions, programmers can avoid saving registers by managing all registers globally. This can significantly reduce overhead that results from the spilling and filling of registers to and from the stack.

Because coding in assembly language is time-consuming, programmers can use the strategy of hand-coding only the most frequently executed DSP kernels; other files can be written in C, using the methods described in Sections Section 2.2 “asm Macros in C”, Section 2.3 “Intrinsics in C”, and Section 2.4 “Fixed-Point Data Types and Operators in C”.

MIPS Technologies provides a MIPS32 DSP Quick Reference Card [3] to enable programmers to quickly learn the MIPS DSP ASE instructions.

2.2 asm Macros in C

GCC supports asm macros that produce DSP instructions directly from C code. The description of asm macros is in the GCC manual at the following URL:

http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Extended-Asm.html

The MIPS DSP ASE instructions can be coded in asm macros in C with two rules.

1. Variables must be mapped to registers via register constraints.

2. Data dependencies must be properly maintained.

The register constraint specifier for the 32 general-purpose registers is `d`. For the four DSP accumulators, the register constraint specifier is `x`. The register constraint can be a number (e.g., 0, 1, 2), which constrains the operand to use the same register.

As an example, for the `dpa.w.ph` instruction, we want to specify:

1. The output register is an accumulator, and

2. the input register uses the same accumulator.

We can use `=x` as the register-constraint specifier for the output (operand 0) and use "`0`" as the specifier for the input register:

```
asm ("dpa.w.ph %q0, %2, %3": "=x" (c) : "0" (c), "d" (a), "d" (b)).
```

The operand formats inside an instruction use "`%`" plus "0", "1", "2", etc. for general-purpose register operands 0, 1, 2, etc., and use "`%q`" plus "0", "1", "2", etc. for accumulator operands 0, 1, 2, etc.. Immediate operands can be entered directly in the code. For instructions that use an immediate number as an operand, programmers can just put the number into the instruction operand.

To prevent the compiler from optimizing the assembly code, use the `asm volatile` compiler directive. Using this directive is very helpful in dealing with dependencies in the DSP control registers, because the DSP control registers are not available as operands in asm macros.

Some asm macro examples are shown below.

```
int f0 (int a)
{
  int c;

  // "c" maps to an output general purpose register (operand 0)
  // "a" maps to an input general purpose registers (operand 1)
  asm ("absq_s.qb %0, %1": "=d" (c) : "d" (a));
  return c;
}

int f1 (int a, int b)
{
  int c;
  // "c" maps to an output general purpose register (operand 0)
  // "a" maps to an input general purpose register (operand 1)
```

```
  // "b" maps to an input general purpose register (operand 2)
  asm ("addq_s.ph %0, %1, %2": "=d" (c) : "d" (a), "d" (b));
  return c;
}

long long f2 (int a, int b)
{
  long long c = 0;
  // "c" maps to an output accumulator (operand 0)
  // "c" maps to an input accumulator (operand 1)
  // "a" maps to an input general purpose register (operand 2)
  // "b" maps to an input general purpose register (operand 3)
  asm ("dpa.w.ph %q0, %2, %3": "=x" (c) : "0" (c), "d" (a), "d" (b));
  return c;
}

int f3 ()
{
  int c;
  // "c" maps to an ouput general purpose register (operand 0)
  asm ("rddsp %0, 63": "=d" (c));
  return c;
}

void f4 (int a)
{
  // "a" maps to an input general purpose register (operand 0)
  asm volatile ("wrdsp %0, 63": : "d" (a));
}
```

Using asm macros in C is easier than writing assembly, because there is no need to allocate registers manually and there is no need to worry about the correctness of the assembly-file format. However, GCC treats each asm macro as a single-cycle instruction, so its scheduling of these DSP ASE instructions is not optimal. To allow the compiler to schedule the MIPS DSP ASE instructions based on latencies, use C-language intrinsics, as described in 2.3 "Intrinsics in C".

2.3 Intrinsics in C

GCC supports intrinsics for the MIPS DSP ASE. The description of these intrinsics is in the GCC manual, which can be downloaded from:

http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/MIPS-DSP-Built_002din-Functions.html

The MIPS DSP ASE intrinsics are also described in the MIPS document *Efficient DSP ASE Programming in C: Tips and Tricks* [4].

SIMD data types (also called "vector types"), in addition to C scalar data types (e.g., char, short, int, long, long long), are required to use the MIPS DSP ASE intrinsics. The description of the SIMD data types is in the GCC manual, which can be downloaded from:

http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Vector-Extensions html

Here are some examples:

```
// Scalar data types
typedef int q31;
typedef int i32;
typedef unsigned int ui32;
typedef long long a64;

// SIMD data types
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef signed char v4q7 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

`v4i8`, `v4q7`, `v2i16`", and "`v2q15`" are SIMD data types that consist of 4, 4, 2, and 2 elements of 8 bits, 8 bits, 16 bits, and 16 bits respectively in a single variable/register.

2.3.1 SIMD Operators

Using SIMD data types is a very powerful technique. Programmers can enjoy the performance improvement from SIMD data types by calling the MIPS DSP ASE intrinsics and/or using generic C operators. For SIMD data types, GCC can map C operators (e.g., +, -, *, /) to hardware instructions directly, so long as the target has the corresponding instructions.

Here are some examples:

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
v4i8 a, b, c;
c = a + b; // GCC will generate addu.qb for MIPS DSP ASE Rev1
c = a - b; // GCC will generate subu.qb for MIPS DSP ASE Rev1

typedef short v2q15 __attribute__ ((vector_size(4)));
v2q15 d, e, f;
f = d + e; // GCC will generate addq.ph for MIPS DSP ASE Rev1
f = d - e; // GCC will generate subq.ph for MIPS DSP ASE Rev1

typedef short v2i16 __attribute__ ((vector_size(4)));
v2i16 x, y, z;
z = x * y; // GCC will generate mul.ph for MIPS DSP ASE Rev2
```

Note that when "`char`" or '`short`" data elements are packed into SIMD data types, the first data must be aligned to 32 bits; otherwise, the unaligned memory accesses generate exceptions and decrease performance.

2.3.2 MIPS DSP ASE Rev1 Intrinsics

The MIPS DSP ASE Rev1 intrinsics are listed below.

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
q31 __builtin_mips_addq_s_w (q31, q31);
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
```

```
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
q31 __builtin_mips_subq_s_w (q31, q31);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);
i32 __builtin_mips_modsub (i32, i32);
i32 __builtin_mips_raddu_w_qb (v4i8);
v2q15 __builtin_mips_absq_s_ph (v2q15);
q31 __builtin_mips_absq_s_w (q31);
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
v2q15 __builtin_mips_precequ_ph_qbl (v4i8);
v2q15 __builtin_mips_precequ_ph_qbr (v4i8);
v2q15 __builtin_mips_precequ_ph_qbla (v4i8);
v2q15 __builtin_mips_precequ_ph_qbra (v4i8);
v2q15 __builtin_mips_preceu_ph_qbl (v4i8);
v2q15 __builtin_mips_preceu_ph_qbr (v4i8);
v2q15 __builtin_mips_preceu_ph_qbla (v4i8);
v2q15 __builtin_mips_preceu_ph_qbra (v4i8);
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_ph (v2q15, i32);
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_s_ph (v2q15, i32);
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_ph (v2q15, i32);
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_r_ph (v2q15, i32);
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
```

```
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);
i32 __builtin_mips_bitrev (i32);
i32 __builtin_mips_insv (i32, i32);
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (i32);
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8);
void __builtin_mips_cmpu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
i32 __builtin_mips_extr_w (a64, imm0_31);
i32 __builtin_mips_extr_w (a64, i32);
i32 __builtin_mips_extr_r_w (a64, imm0_31);
i32 __builtin_mips_extr_s_h (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, imm0_31);
i32 __builtin_mips_extr_rs_w (a64, i32);
i32 __builtin_mips_extr_s_h (a64, imm0_31);
i32 __builtin_mips_extr_r_w (a64, i32);
i32 __builtin_mips_extp (a64, imm0_31);
i32 __builtin_mips_extp (a64, i32);
i32 __builtin_mips_extpdp (a64, imm0_31);
i32 __builtin_mips_extpdp (a64, i32);
a64 __builtin_mips_shilo (a64, imm_n32_31);
a64 __builtin_mips_shilo (a64, i32);
a64 __builtin_mips_mthlip (a64, i32);
void __builtin_mips_wrdsp (i32, imm0_63);
i32 __builtin_mips_rddsp (imm0_63);
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
i32 __builtin_mips_bposge32 (void);
NOTE: The following six intrinsics are moved from Rev2 to Rev1 in GCC 4.6 and
higher.
a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, i32, i32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, i32, i32);
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (i32, i32);
```

2.3.3 MIPS DSP ASE Rev2 Intrinsics

The MIPS DSP ASE Rev2 intrinsics are listed below.

```
v4q7 __builtin_mips_absq_s_qb (v4q7);
v2i16 __builtin_mips_addu_ph (v2i16, v2i16);
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);
```

```
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);
i32 __builtin_mips_append (i32, i32, imm0_31);
i32 __builtin_mips_balign (i32, i32, imm0_3);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);
v2i16 __builtin_mips_mul_ph (v2i16, v2i16);
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);
q31 __builtin_mips_mulq_rs_w (q31, q31);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);
q31 __builtin_mips_mulq_s_w (q31, q31);
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);
i32 __builtin_mips_prepend (i32, i32, imm0_31);
v4i8 __builtin_mips_shra_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_qb (v4i8, i32);
v4i8 __builtin_mips_shra_r_qb (v4i8, i32);
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15);
v2i16 __builtin_mips_shrl_ph (v2i16, i32);
v2i16 __builtin_mips_subu_ph (v2i16, v2i16);
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);
```

2.3.4 Optimization Steps

Here are the steps to optimize a program using the MIPS DSP ASE intrinsics.

Step 1. Examine the underlying data types that the program operates. How many bits are there in a single data element? Can multiple data elements be packed into a 32-bit register for SIMD data types?

Step 2. After selecting the data types, refer to the list of MIPS DSP ASE intrinsics for special functions: addition, subtraction, shift left/right, multiplication, multiplication and addition/subtraction, expansion, comparison, picking elements, absolute value, precision reduction/expansion, extraction, branch control, circular buffer indexing, insertion.

There are intrinsics for each SIMD data type and for conversions between these data types.

The advantage of using the MIPS DSP ASE intrinsics is that GCC can schedule DSP instructions based on their latencies, and GCC can allocate registers for all variables. This increases the programmer's productivity, allowing him to focus on algorithm design and code optimization instead of low-level details such as register maintenance.

2.4 Fixed-Point Data Types and Operators in C

The MIPS DSP ASE is the only processor architecture that supports fixed-point data types in a general-purpose processor. Fixed-point data types and operators in C are supported by GCC, as described in the GCC manual:

http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Fixed_002dPoint.html

Fixed-point data types and operators are GCC extensions based on the N1169 draft of ISO/IEC DTR 18037, which supports type checking on fixed-point data types and can process fixed-point data types and calculations inside the compiler.

2.4.1 Fixed-Point Data Types

For MIPS32 systems, the fixed-point data types listed below are supported with specific formats. In the formats shown, "`s`" is the sign bit for a signed type (there is no sign bit for an unsigned type). the period character, "." is the specifier that separates the integral part and the fractional part, and the numeric digits represent the number of bits in the integral part or in the fractional part.

```
short _Fract; // s.7 (Q7)
_Fract; // s.15 (Q15)
long _Fract; // s.31 (Q31)
long long _Fract; // s.63 (Q63)
short _Accum; // s8.7 (Q8.7)
_Accum; // s16.15 (Q16.15)
long _Accum; // s32.31 (Q32.31)
long long _Accum; // s32.31 (Q32.31)
unsigned short _Fract; // .8
unsigned _Fract; // .16
unsigned long _Fract; // .32
unsigned long long _Fract; // .64
unsigned short _Accum; // 8.8
unsigned _Accum; // 16.16
unsigned long _Accum; // 32.32
unsigned long long _Accum; // 32.32
```

The type modifier "`_Sat`" can be added to data-type declarations to specify saturation output behavior, which is commonly used in DSP applications. The saturating fixed-point data types for 32-bit MIPS systems are listed below.

```
_Sat short _Fract; // s.7 (Q7)
_Sat _Fract; // s.15 (Q15)
_Sat long _Fract; // s.31 (Q31)
_Sat long long _Fract; // s.63 (Q63)
_Sat short _Accum; // s8.7 (Q8.7)
_Sat _Accum; // s16.15 (Q16.15)
_Sat long _Accum; // s32.31 (Q32.31)
```

```
_Sat long long _Accum; // s32.31 (Q32.31)
_Sat unsigned short _Fract; // .8
_Sat unsigned _Fract; // .16
_Sat unsigned long _Fract; // .32
_Sat unsigned long long _Fract; // .64
_Sat unsigned short _Accum; // 8.8
_Sat unsigned _Accum; // 16.16
_Sat unsigned long _Accum; // 32.32
_Sat unsigned long long _Accum; // 32.32
```

2.4.2 Fixed-Point Operators

The fixed-point operators include unary operators: ++, −−, +, −, !, binary operators: +, -, *, /, <<, >>, +=, -=, *=, /=, <<=, >>=, and comparison operators: <, <=, >=, >, ==, !=. All other operators are invalid on fixed-point data types.

The following examples show how GCC maps operators to the MIPS DSP ASE instructions for scalar types.

```
short _Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

_Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

long _Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

unsigned short _Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

unsigned _Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

unsigned long _Fract a, b, c;
c = a + b; // addu
c = a - b; // subu

short _Accum a, b, c;
c = a + b; // addu
c = a - b; // subu

_Accum a, b, c;
c = a + b; // addu
c = a - b; // subu

unsigned short _Accum a, b, c;
c = a + b; // addu
c = a - b; // subu

unsigned _Accum a, b, c;
c = a + b; // addu
```

```
c = a - b; // subu

_Sat unsigned short _Fract a, b, c;
c = a + b; // addu_s.qb
c = a - b; // subu_s.qb

_Sat unsigned _Fract a, b, c;
c = a + b; // addu_s.ph
c = a - b; // subu_s.ph

_Sat unsigned short _Accum a, b, c;
c = a + b; // addu_s.ph
c = a - b; // subu_s.ph

_Sat _Fract a, b, c;
c = a + b; // addu_s.ph
c = a - b; // subu_s.ph
c = a * b; // mulq_rs.ph

_Sat long _Fract a, b, c;
c = a + b; // addq_s.w
c = a - b; // subq_s.w
c = a * b; // mulq_rs.w

_Sat short _Accum a, b, c;
c = a + b; // addq_s.ph
c = a - b; // subq_s.ph

_Sat _Accum a, b, c;
c = a + b; // addq_s.w
c = a - b; // subq_s.w
```

As mentioned in Section [2.3], SIMD data types are powerful, and they can be applied to fixed-point data types as well. The following examples show how GCC maps operators to the MIPS DSP ASE instructions for SIMD data types.

```
typedef _Sat unsigned short _Fract sat_v4uqq __attribute__ ((vector_size(4)));
typedef _Sat unsigned _Fract sat_v2uhq __attribute__ ((vector_size(4)));
typedef _Sat unsigned short _Accum sat_v2uha __attribute__ ((vector_size(4)));
typedef _Sat _Fract sat_v2hq __attribute__ ((vector_size(4)));
typedef _Sat short _Accum sat_v2ha __attribute__ ((vector_size(4)));

sat_v2hq a, b, c;
c = a + b; // addq_s.ph
c = a - b; // subq_s.ph
c = a * b; // mulq_rs.ph

sat_v2ha a, b, c;
c = a + b; // addq_s.ph
c = a - b; // subq_s.ph

sat_v4uqq a, b, c;
c = a + b; // addu_s.qb
c = a - b; // subu_s.qb

sat_v2uhq a, b, c;
c = a + b; // addu_s.ph
c = a - b; // subu_s.ph
```

```
sat_v2uha a, b, c;
c = a + b; // addu_s.ph
c = a - b; // subu_s.ph
```

2.4.3 Optimization Steps

Here are the steps to optimize a program using fixed-point data types and operators:

Step 1. Examine the underlying data types that the program operates. Can the fixed-point data types be used? How many bits are there in the integral part and in the fractional part of a type? Is the type signed or unsigned? Does the type need saturation (_Sat)? Can SIMD data types be used to improve the performance?

Step 2. After selecting the data types, use the supported C operators to operate on fixed-point scalar or SIMD variables.

The advantage of the fixed-point method is that it is easy to write a DSP program using fixed-point data types and operators. Programmers can use integer, fixed-point, and floating-point data types in the same way. There is no need to memorize the names of intrinsics. However, not all fixed-point operators can be mapped to MIPS DSP ASE instructions, and when they can't be mapped, GCC calls an emulation function to perform the calculation.

Note that the fixed-point method is orthogonal to the intrinsic method (Section 2.3 "Intrinsics in C"). Therefore, programmers may use both the fixed-point method for ease of programming and the intrinsic method for the best performance, by invoking specific MIPS DSP ASE instructions.

2.5 Auto-Vectorization

GCC supports auto-vectorization for loops via the optimization option: `-ftree-vectorize`. This option is enabled by default when the `-O3` option is selected. The advantage of auto-vectorization is that the compiler can recognize scalar variables (which can be integer, fixed-point, or floating-point types) in order to utilize SIMD instructions automatically. In the ideal case, when auto-vectorization is used, there is no need to use SIMD variables explicitly.

To see how many loops are auto-vectorized by GCC, `-fdump-tree-vect-stats` can be used to generate an optimization log when compiling code. For example:

```
# cat add8.c
unsigned char a[32], b[32], c[32];

void add8()
{
  int i;
  for (i = 0; i < 32; i++)
  {
    c[i] = a[i] + b[i];
  }
}

# mips-sde-elf-gcc -ftree-vectorize -O2 -mdspr2 add8.c -c -fdump-tree-vect-stats

# grep vectorized add8.c.103t.vect
```

```
add8.c:3: note: vectorized 1 loops in function.

# mips-sde-elf-objdump -d add8.o

add8.o:     file format elf32-tradbigmips


Disassembly of section .text:

00000000 <add8>:
   0:   3c090000        lui     t1,0x0
   4:   3c080000        lui     t0,0x0
   8:   3c070000        lui     a3,0x0
   c:   25290000        addiu   t1,t1,0
  10:   25080000        addiu   t0,t0,0
  14:   24e70000        addiu   a3,a3,0
  18:   00001021        move    v0,zero
  1c:   24060020        li      a2,32
  20:   01022021        addu    a0,t0,v0
  24:   00e21821        addu    v1,a3,v0
  28:   8c840000        lw      a0,0(a0)
  2c:   8c630000        lw      v1,0(v1)
  30:   01222821        addu    a1,t1,v0
  34:   24420004        addiu   v0,v0,4
  38:   7c831810        addu.qb v1,a0,v1
  3c:   1446fff8        bne     v0,a2,20 <add8+0x20>
  40:   aca30000        sw      v1,0(a1)
  44:   03e00008        jr      ra
  48:   00000000        nop

# cat add16.c
_Sat _Fract a[12], b[12], c[12];

void add16()
{
  int i;
  for (i = 0; i < 12; i++)
  {
    c[i] = a[i] + b[i];
  }
}

# mips-sde-elf-gcc -O2 -ftree-vectorize -mdspr2 -c add16.c -fdump-tree-vect-stats

# grep vectorized add16.c.103t.vect
add16.c:3: note: vectorized 1 loops in function.

# mips-sde-elf-objdump -d add16.o

add16.o:     file format elf32-tradbigmips


Disassembly of section .text:

00000000 <add16>:
   0:   3c090000        lui     t1,0x0
   4:   3c080000        lui     t0,0x0
   8:   3c070000        lui     a3,0x0
```

```
 c:   25290000        addiu   t1,t1,0
10:   25080000        addiu   t0,t0,0
14:   24e70000        addiu   a3,a3,0
18:   00001021        move    v0,zero
1c:   24060018        li      a2,24
20:   01022021        addu    a0,t0,v0
24:   00e21821        addu    v1,a3,v0
28:   8c840000        lw      a0,0(a0)
2c:   8c630000        lw      v1,0(v1)
30:   01222821        addu    a1,t1,v0
34:   24420004        addiu   v0,v0,4
38:   7c831b90        addq_s.ph       v1,a0,v1
3c:   1446fff8        bne     v0,a2,20 <add16+0x20>
40:   aca30000        sw      v1,0(a1)
44:   03e00008        jr      ra
48:   00000000        nop
```

In `add8.c`, elements in two arrays of `unsigned char` are added together. GCC automatically generates the code for `addu.qb` to add four elements at a time. In `add16.c`, elements in two arrays of `_Sat _Fract` are added together. GCC automatically generates the code for `addq_s.ph` for saturating addition (two elements at a time).

For existing C code, programmers can try auto-vectorization without any modifications to see if loops can be vectorized by GCC. In some cases, if the loop body is too complex, GCC will not be able to auto-vectorize the loop; in this case, the code can be rewritten and loops restructured.

3 Conclusion

Choosing the method of utilizing the MIPS DSP ASE depends on the effort programmers want to make and the performance goals they intend to reach.

Writing assembly code is the most time-consuming method but can obtain the highest performance. The least time-consuming method, which requires no source code modification, is to use GCC's auto-vectorization. In between, programmers can use asm macros, intrinsics, or fixed-point data types and operators in the C language. Using asm macros is a quick method to utilize the MIPS DSP ASE instructions. Using intrinsics is a better method than using asm macros, because GCC can schedule the MIPS DSP ASE instructions based on latencies. Using fixed-point data types and operators is an easier method, because there is no need to memorize the names of the MIPS DSP ASE intrinsics.

For fast time-to-market, using fixed-point data types and operators or just applying auto-vectorization may be a good choice. For high performance, design the algorithm and optimize the code in assembly or in C by using asm macros and MIPS DSP ASE intrinsics.

4 References

1. MIPS32® Architecture for Programmers Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture
   MIPS Document: MD00374

2. MIPS® Architecture Reference Manual Volume IV-e: The MIPS® DSP Application-Specific Extension to the microMIPS32™ Architecture 762

3. MIPS32® DSP ASE Instruction Set Quick Reference
   MIPS Document: MD00566

4. Efficient DSP ASE Programming in C: Tips and Tricks
   MIPS Document: MD00485

5. Effective DSP Programming using MIPS® DSP Application Specific Extensions
   MIPS Document: MD00475

6. MIPS® DSP Library Application Programmer Interface
   MIPS Document: MD00358

7. MIPS® DSP Library Release Notes
   MIPS Document: MD00364

8. MIPS® DSP Library Reference Manual: DSP Module
   MIPS Document: MD00472

9. JPEG Decoder Optimization using MIPS® DSP Application Specific Extensions
   MIPS Document: MD00483

10. Optimizing G.729AB using MIPS® DSP Application Specific Extensions
    MIPS Document: MD004484

11. MD00495 MP3 Decoder Optimization using MIPS® DSP Application Specific Extensions
    MIPS Document: MD00495

12. H.263 Decoder Optimization using MIPS® DSP Application Specific Extensions
    MIPS Document: MD00533

13. Programming the MIPS(R) 74K™ Core Family for DSP Applications
    MIPS Document: MD00544

14. *See MIPS Run*, Second Edition, Dominic Sweetman
    Morgan Kaufmann Publishers

15. GCC, the GNU Compiler Collection
    http://gcc.gnu.org

16. GNU Binutils
    http://www.gnu.org/software/binutils/

CłØŜ aŜűK2Řă 2ŧ ¦ůłfůÎłyƎ űKŜ aʟt{† 5{t !{9Σ wŜØłăł2yϒ лм¢лм