Efficient DSP ASE Programming in C: Tips and Tricks

This paper is designed to help DSP programmers migrate from hand optimized assembly programming to pure C coding by using the MIPS32 DSP ASE (Architecture Specific Extension) and GCC, a C compiler with optimizations for MIPS architecture and cores. Various tips and tricks for writing efficient C code are presented, including the creation of variable types to exploit "Single Instruction Multiple Data" features in the ASE, and use of compiler "Intrinsics" to provide direct access to DSP instructions. C code written with these tricks is quicker to develop, easier to maintain, and more portable than hand written assembly code, with little to no reduction in speed. DSP programmers should exploit these tips to enjoy high performance and faster time-to-market using the MIPS32 DSP ASE.

Document Number: MD00485
Revision 01.22
June 7, 2011

Table of Contents

Section 1: Overview 5

Section 2: Introduction to the MIPS32® DSP ASE 6
- 2.1: Enabling the DSP ASE in the Compiler 6
- 2.2: Data Types 6
- 2.3: Initialization of Q15 and Q31 Variables 7
- 2.4: Initialization of SIMD Variables 7
- 2.5: Accessing Elements in SIMD Variables 8
- 2.6: C Operators 9
- 2.7: C Intrinsics for the MIPS32 DSP ASE 10
- 2.8: DSP Control Register 12

Section 3: Introduction to C-Based Intrinsics for the MIPS32® DSP ASE 13
- 3.1: Intrinsics for Instructions that Access and Use the DSP Control Register 13
  - 3.1.1: Read/Write the DSP Control Register 13
  - 3.1.2: Branch on Greater Than or Equal to 32 in POS 14
- 3.2: Using Intrinsics for Signed and Unsigned 8-bit Integers 14
  - 3.2.1: Unsigned Add/Subtract with Optional Saturation 15
  - 3.2.2: Unsigned Reduction Add 15
  - 3.2.3: Shift Left/Right Logical 15
  - 3.2.4: Dot Product with Accumulate/Subtract 15
  - 3.2.5: Replicate a Fixed Byte Value into SIMD Elements 16
  - 3.2.6: Compare Unsigned (DSPR1/DSPR2) 16
  - 3.2.7: Pick Based on Condition Code Bits 16
  - 3.2.8: Find Absolute Value (DSPR2) 17
  - 3.2.9: Unsigned Add and Right Shift to Halve Results with Optional Rounding (DSPR2) 17
  - 3.2.10: Shift Right Arithmetic with Optional Rounding (DSPR2) 17
  - 3.2.11: Unsigned Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2) 17
- 3.3: Using Intrinsics for Q15 Data Type 18
  - 3.3.1: Add/Subtract with Optional Saturation 18
  - 3.3.2: Find Absolute Value 18
  - 3.3.3: Shift Left Logical with Optional Saturation 18
  - 3.3.4: Shift Right Arithmetic with Optional Rounding 19
  - 3.3.5: Multiply with Rounding and Saturation (Q15 x Q15 => Q15) 19
  - 3.3.6: Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31) 19
  - 3.3.7: Multiply and Subtract and Accumulate (Q15 x Q15 => Q32.31) 19
  - 3.3.8: Multiply with Accumulate a Single Element (Q15 x Q15 => Q31) 20
  - 3.3.9: Multiply Vector Fractional Left/Right Half-Words to Expanded Width Product with Saturation (Q15 x Q15 => Q31) 20
  - 3.3.10: Replicate a Fixed Half-word into Elements 21
  - 3.3.11: Compare 21
  - 3.3.12: Pick Based on Condition Code Bits 21
  - 3.3.13: Pack from the Right and Left 21
  - 3.3.14: Multiply with Saturation (Q15 x Q15 => Q15) (DSPR2) 21
  - 3.3.15: Add and Right Shift to Halve Results with Optional Rounding (DSPR2) 22
  - 3.3.16: Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2) 22
  - 3.3.17: Cross Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31) (DSPR2) 22

3.4: Using Intrinsics for Q31 Data Type 23
3.4.1: Add/Subtract with Saturation 23
3.4.2: Find Absolute Value 23
3.4.3: Shift Left Logical with Saturation 23
3.4.4: Shift Right Arithmetic with Rounding 23
3.4.5: Dot Product with Accumulate/Subtract (Q31 x Q31 => Q63) 24
3.4.6: Multiply with Rounding and Saturation (Q31 x Q31 => Q31) (DSPR2) 24
3.4.7: Multiply with Saturation (Q31 x Q31 => Q31) (DSPR2) 24
3.4.8: Add and Right Shift to Halve Results with Optional Rounding (DSPR2) 24
3.4.9: Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2) 24
3.5: Using Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers 25
3.5.1: Precision Reduce Four Fractional Half-words to Four Bytes 25
3.5.2: Precision Reduce Unsigned Four Fractional Half-words to Four Bytes with Saturation 25
3.5.3: Precision Expand Two Unsigned Bytes to Fractional Half-word Values 25
3.5.4: Precision Expand Two Unsigned Bytes to Unsigned Integer Half-words 26
3.5.5: Multiply Unsigned Vector Left/Right Bytes with Half-Words to Half Word Products with Saturation (Int8 x Q15 => Q15) 26
3.5.6: Precision Reduce Four Integer Half-words to Four Bytes (DSPR2) 26
3.6: Using Intrinsics for Mixed Data Types: Q15 and Q31 26
3.6.1: Precision Reduce Two Fractional Words to Two Half-Words 26
3.6.2: Precision Reduce Two Fractional Words to Two Half-Words with Rounding and Saturation 27
3.6.3: Precision Expand a Fractional Half-word to a Fractional Word Value 27
3.7: Using Intrinsics for 64-bit Accumulators 27
3.7.1: Extract a Value with Right Shift 27
3.7.2: Extract Half-word with Right Shift and Saturate 27
3.7.3: Extract Bit from an Arbitrary Position 28
3.7.4: Extract Bit from an Arbitrary Position and Decrement POS 28
3.7.5: Shift an Accumulator Value 28
3.7.6: Copy the LO to HI and a Value to LO and Increment POS by 32 29
3.8: Using Intrinsics for 32-bit Integers 29
3.8.1: Add and Set Carry/Add with Carry 29
3.8.2: Modular Subtraction on an Index Value 29
3.8.3: Bit Reverse a Half-word 29
3.8.4: Insert Bit Field Variable 30
3.8.5: Load Unsigned Byte/Halfword/Word Indexed 30
3.8.6: Signed Multiply and Add 30
3.8.7: Unsigned Mulitply and Add 31
3.8.8: Signed Multiply and Subtract 31
3.8.9: Unsigned Multiply and Subtract 31
3.8.10: Signed Multiply 31
3.8.11: Unsigned Multiply 31
3.8.12: Left Shift and Append Bits (DSPR2) 31
3.8.13: Byte Align Contents from Two Registers (DSPR2) 32
3.8.14: Right Shift and Prepend Bits (DSPR2) 32
3.9: Using Intrinsics for 16-bit Integers 32
3.9.1: Unsigned Add/Subtract with Optional Saturation (DSPR2) 32
3.9.2: Dot Product with Accumulate/Subtract (DSPR2) 32
3.9.3: Multiply with Optional Saturation (DSPR2) 33
3.9.4: Multiply and Subtract and Accumulate (DSPR2) 33
3.9.5: Shift Right Logical (DSPR2) 33
3.9.6: Cross Dot Product with Accumulate/Subtract (DSPR2) 33
3.10: Using Intrinsics for Mixed Data Types: 16-bit and 32-bit Integers 34
3.10.1: Precision Reduce Two Integer Words to Halfwords After a Right Shift with Optional Rounding

(DSPR2) ... 34

Section 4: Tips for Efficient Code ... 34
4.1: Compiler Options for Optimization and CPUs ... 34
4.2: The Use of Intrinsics versus Asm Macros ... 35
4.3: Using Accumulators ... 35
4.4: Multiply “32-bit * 32-bit = 64-bit” ... 36
4.5: Multiply and Add “32-bit * 32-bit + 64-bit = 64-bit” ... 36
4.6: Array Alignment and Data Layout ... 37
4.7: GP-Relative Addressing ... 37

Section 5: Advanced Topics and Some Complex Usage ... 38
5.1: Fixed Registers and Register Variables ... 38
5.2: Conditional Moves ... 42

Section 6: A Programming Example ... 43
6.1: The FIR Filter in Traditional C ... 43
6.2: The FIR Filter in Hand-Tuned Assembly ... 43
6.3: The FIR Filter in Efficient C ... 44

Section 7: MIPS32 DSP Intrinsics ... 45
7.1: Intrinsics for DSP Control Register ... 45
7.2: Intrinsics for Signed and Unsigned 8-bit Integers ... 45
7.3: Intrinsics for Q15 ... 46
7.4: Intrinsics for Q31 ... 46
7.5: Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers ... 47
7.6: Intrinsics for Mixed Data Types: Q15 and Q31 ... 47
7.7: Intrinsics for 64-bit Accumulators ... 47
7.8: Intrinsics for 32-bit Integers ... 47
7.9: Intrinsics for 16-bit Integers ... 48
7.10: Intrinsics for Mixed DataTypes: 16-bit and 32-bit Integers ... 48

Section 8: Experimental Results and Summary ... 48

Section 9: References ... 49

1 Overview

The MIPS® DSP ASE (Application Specific Extension) to the MIPS32 architecture was introduced by MIPS Technologies to optimize the performance of signal processing and multimedia applications running on MIPS core processors. The DSP ASE is a set of new instructions and new architectural state, with computational support for fractional data types, SIMD, saturation, and other operations commonly used in DSP applications.

Typical DSP applications include certain loops or kernels that take a large percentage of the execution time. Because of the sensitivity of DSP application performance to these kernels, programmers have tended to write these functions in assembly, hand-scheduling the code for optimal pipeline scheduling. Another key reason for assembly programming has been that compilers for Digital Signal Processors have not been very effective in back-end code scheduling. But with the introduction of RISC-based DSP extensions in MIPS cores, we can now take advantage of compilers like GCC that generate optimized code for RISC processors. The use of such compilers reduces, and may even eliminate, the need to write assembly code. However, to obtain the best optimizations, a particular coding style and usage must be followed, as explained in this paper.

The GCC Compiler 6.03.00-rc3, based on FSF (Free Software Foundation) GCC 3.4, was used for the examples in this paper. Programmers should use the newest compilers available for MIPS cores to ensure that the DSP ASE is supported and to enable the highest performance instruction scheduling. Such compilers include the SG++ toolchains for MIPS (Spring 2008 and newer) from CodeSourcery. Throughout the rest of the document, the term “GCC compiler” refers to a compiler that incorporates the DSP ASE intrinsics, such as the GCC compiler from CodeSourcery.

Using a high-level language such as C to develop applications has many advantages:

1. C programmers do not have to manually allocate registers—the compiler can allocate registers for variables.

2. C programmers do not have to manually schedule instructions—the compiler can schedule instructions based on given latency information for specific CPU pipelines.

3. C programmers do not have to consider function calling conventions when writing modular programs—the compiler saves and restores registers at entries and exits of functions per a pre-defined convention.

4. C programmers can declare SIMD variables and use generic C operators or intrinsics (built-in functions) to manage SIMD variables in order to achieve parallelism.

5. Development and debug time is shortened when using a high-level language.

6. Applications are more maintainable when written in a high-level language.

The rest of this paper is organized as follows:

- Section 2 presents enhanced C programming features that include data types (fixed-point, SIMD, and accumulators), union types, operators, and C-callable intrinsics for the MIPS32 DSP ASE.

- Section 3 introduces the MIPS32 DSP intrinsics based on categories.

- Section 4 lists several tips on how to write efficient C code, and includes its corresponding compiler-generated assembly code.

- Section 5 explains how to treat a register as fixed (or global) and how to use asm macros to generate conditional move instructions.

- Section 6 presents a programming example.
- Section 7 lists all provided intrinsics supported by the GCC Compiler.
- Section 8 compares the performance of several DSP kernels written in hand-tuned assembly code and in efficient C code.

2 Introduction to the MIPS32® DSP ASE

The MIPS32 DSP ASE includes new instructions to improve the performance of DSP and media applications. Many of these new instructions operate on 8-bit unsigned integer data and signed Q15 or Q31 fractional data. In addition, many instructions saturate their results to a 16-bit or 32-bit value as appropriate.

The 8-bit unsigned integer data and Q15 fractional data are packed in a single 32-bit register, and the new instructions operate simultaneously on the multiple data in the register in Single Instruction, Multiple Data (SIMD) fashion. This provides computation parallelism for increased application performance. For detailed information about the MIPS32 DSP ASE instructions, please refer to MIPS32® Architecture for Programmers Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture (MD00374).

2.1 Enabling the DSP ASE in the Compiler

To enable the MIPS32 DSP ASE in the GCC compiler, the “-mdsp” or “-mdsp2” command-line option is required. In addition to “-mdsp” or “-mdsp2”, “-mips32r2” is recommended for better performance in accessing elements in SIMD variables, because this allows the use of the MIPS32 Release 2 instruction “INS” for more efficient code. Note that MIPS32 24KE and 34K cores that implement the DSP ASE revision 1 require the "-mdsp" option; 74K and M14KE cores that implement the DSP ASE revision 2 require the "-mdspr2" option. For the M14KE core, DSP instructions can be assembled into MIPS opcode or microMIPS opcode (with the "-mmicromips" command-line option).

For the SG++ Bare-Iron compiler:

```
mips-sde-elf-gcc -c -O2 -mtune=2k4ec -mips32r2 -mdsp test.c
```

2.2 Data Types

In the GCC compiler, the Q15 data type is represented by 16-bit integer data type (short), and the Q31 data type is represented by 32-bit integer data type (int). Typedefs can be created for Q15 and Q31 as follows:

```
typedef short q15;
typedef int q31;
```

The MIPS32 DSP ASE implements four 64-bit accumulators which can be represented by a “long long” data type.

```
typedef long long a64;
```

To declare SIMD data types, typedefs with special vector_size attributes are required. For example,

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

where “v4i8” defines a SIMD data type containing four 8-bit integer data. “v2q15” defines a type containing two Q15 fractional data (which is the same as two 16-bit integer data).

2.3 Initialization of Q15 and Q31 Variables

To initialize Q15 variables, programmers can multiply the fractional value (e.g., 0.1234) by 32768.0. To initialize Q31 variables, programmers can multiply the fractional value by 2147483648.0.

```
Ex: /* Q15 Example  */
typedef short q15;
q15 a = 0.1234 * 32768.0;
/* --------------------------------------------------------- */

Ex: /* Q31 Example  */typedef int q31;
q31 b = 0.2468 * 2147483648.0;
```

2.4 Initialization of SIMD Variables

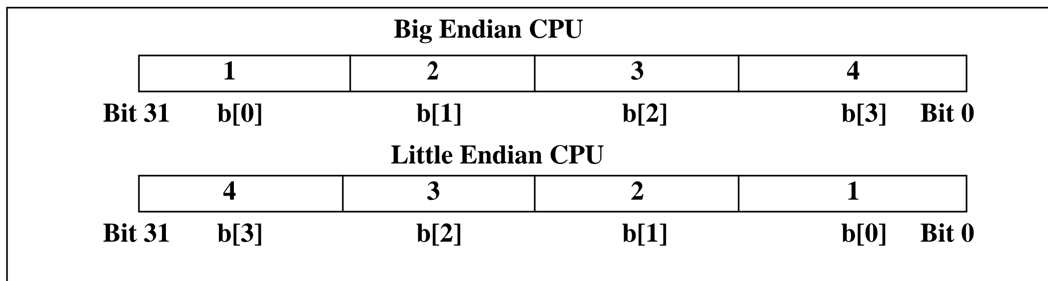
To initialize SIMD variables is similar to initializing aggregate data. The following examples show how to initialize SIMD variables.

```
Ex: /* v4i8 Example  */
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};
/* --------------------------------------------------------- */

Ex: /* v2q15 Example  */
v2q15 a = {0x0fcb, 0x3a75};
v2q15 b;
b = (v2q15) {0.1234 * 32768.0, 0.4567 * 32768.0};
```

Note that CPU endianness affects the ordering of data stored in a register. In a Big Endian CPU, data is stored from the left-to-right location of a register. In a Little Endian CPU, data is stored from the right-to-left location of a register. For the example of v4i8 a = {1, 2, 3, 4}, a Big Endian CPU stores 1, 2, 3, and 4 from the left-to-right location, but a Little Endian CPU stores 1, 2, 3, and 4 from the right-to-left location as shown in Figure 1.

Figure 1 Register Values for v4i8 a = {1, 2, 3, 4} in Big Endian and Little Endian CPUs



Most arithmetic operations will simply work on the SIMD operands in the register irrespective of endianness. But the programmer must beware of such instructions in the DSP ASE that directly refer to the left or right portions of a GPR. For example, MAQ_SA.W.PHL.

2.5 Accessing Elements in SIMD Variables

The use of SIMD variables enables operations on multiple data in parallel. However, in certain situations, programmers need to access elements inside a SIMD variable. This can be done by using a union type that unites a SIMD type and an array of a basic type as follows.

```
typedef union
{
  v4i8 a;
  unsigned char b[4];
} v4i8_union;

typedef short q15;
typedef union
{
  v2q15 a;
  q15 b[2];
} v2q15_union;
```

As shown in Figure 1 for a v4i8 variable, b[0] is used in both big-endian and Little Endian CPUs to access the first element in the variable. However, b[0] is stored in the left-most position in a Big Endian CPU, but it is stored in the right-most position in a little-endian CPU. The following examples show how to extract or assign elements.

```
Ex: /* v4i8 Example  */
v4i8 i;
unsigned char j, k, l, m;
v4i8_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];
l = temp.b[2];
m = temp.b[3];

/* Assume we want to assign j, k, l, m to i.  */
temp.b[0] = j;
```

```
temp.b[1] = k;
temp.b[2] = l;
temp.b[3] = m;
i = temp.a;
/* -------------------------------------------------------- */

Ex: /* v2q15 Example  */
v2q15 i;
q15 j, k;
v2q15_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];

/* Assume we want to assign j, k to i.  */
temp.b[0] = j;
temp.b[1] = k;
i = temp.a;
```

2.6 C Operators

Addition and subtraction on fractional data are similar to addition and subtraction with integer data, but multiplication requires a post-multiply shift to align the resulting values appropriately. Because fractional data uses integer data types in the GCC compiler, users must be very cautious when applying operators upon fractional data. The GCC compiler accepts all operators upon Q15 and Q31 data, but only addition and subtraction generate the expected results for Q15 and Q31. Note that operators other than "+" and "-" upon Q15 and Q31 are treated as integer arithmetic.

```
Ex: /* Q15 Example  */
typedef short q15;
q15 i, j, k, l;
i = k + l;
j = k - l;
/* -------------------------------------------------------- */

Ex: /* Q31 Example  */
typedef int q31;
q31 i, j, k, l;
i = k + l;
j = k - l;
```

Certain C operators can be applied to SIMD variables. They are +, -, *, /, unary minus, ^, |, &, ~. The MIPS32 DSP ASE provides SIMD addition and subtraction instructions for v4i8 and v2q15, allowing the GCC compiler to generate appropriate instructions for addition and subtraction of v4i8 and v2q15 variables. For other operators, the GCC compiler synthesizes a sequence of instructions. The examples here show compiler-generated SIMD instructions when the appropriate operator is applied to SIMD variables.

```
Ex: /* v4i8 Addition  */
v4i8 test1 (v4i8 a, v4i8 b)
{
  return a + b;
}

# Generated Assembly
```

```
test1:
  j       $31
  addu.qb $2, $4, $5
# ----------------------------------------------------------

Ex: /* v4i8 Subtraction  */
v4i8 test2 (v4i8 a, v4i8 b)
{
  return a - b;
}

# Generated Assembly
test2:
  j       $31
  subu.qb $2, $4, $5
# ----------------------------------------------------------

Ex: /* v2q15 Addition  */
v2q15 test3 (v2q15 a, v2q15 b)
{
  return a + b;
}

# Generated Assembly
test3:
  j       $31
  addq.ph $2, $4, $5
# ----------------------------------------------------------

Ex: /* v2q15 Subtraction  */
v2q15 test4 (v2q15 a, v2q15 b)
{
  return a - b;
}

# Generated Assembly
test4:
  j       $31
  subq.ph $2, $4, $5
```

In situations where special integer and fractional calculations are needed and the compiler cannot generate them automatically, C intrinsics can be directly used by the programmer as described in the next section.

2.7 C Intrinsics for the MIPS32 DSP ASE

Intrinsics are very similar to function calls in syntax. Programmers need to pass parameters to intrinsics, and intrinsics return results to variables. The difference between intrinsics and functions is that the compiler directly maps intrinsics to instructions for better performance. Intrinsics for the MIPS32 DSP ASE use the following data types:

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
typedef int q31;
typedef int i32;
typedef long long a64;
typedef unsigned int ui32;
```

NOTE: “q31” and “i32” are actually the same as “int”, but the intrinsic that accepts “q31” processes data as Q31 fractional data, and the intrinsic that accepts “i32” processes data as 32-bit integer data.

NOTE: “a64” is the same as “long long”, but the compiler allocates “a64” variables to accumulators ($ac0, $ac1, $ac2, $ac3) ready to be operated on relevant DSP instructions.

The list of all intrinsics can be found in the MIPS32 DSP Intrinsics, and Section 3 will introduce each MIPS32 DSP intrinsic. Programmers should be familiar with the semantics of all MIPS32 DSP instructions so that the corresponding intrinsic can be used appropriately in C programs to achieve better performance.

One example of an intrinsic for the ADDQ.PH instruction is “v2q15 __builtin_mips_addq_ph (v2q15, v2q15).” Two v2q15 variables are required to be passed to “__builtin_mips_addq_ph” and one v2q15 variable is needed to receive the returned result from this intrinsic. The following C code demonstrates the usage of “__builtin_mips_addq_ph”.

```
Ex:
v2q15 test5 ()
{
  v2q15 a, b, c;
  a = (v2q15) {0.12 * 32768.0, 0.34 * 32768.0};
  b = (v2q15) {0.56 * 32768.0, 0.78 * 32768.0};
  c = __builtin_mips_addq_ph (a, b);
  return c;
}

# Generated Assembly
        .file   1 "test5.c"
        .section .mdebug.abi32
        .previous
        .section        .rodata.cst4,"aM",@progbits,4
        .align  2
.LC0:
        .half   3921
        .half   11141
        .align  2
.LC1:
        .half   18299
        .half   25559
        .text
        .align  2
        .align  3
        .globl  test5
        .set    nomips16
        .ent    test5
test5:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        lui     $5,%hi(.LC0)
        lui     $4,%hi(.LC1)
        lw      $2,%lo(.LC0)($5)
        lw      $3,%lo(.LC1)($4)
        j       $31
        addq.ph $2,$2,$3
```

```
        .set    macro
        .set    reorder
        .end    test5
        .ident  "GCC: (GNU) 3.4.4 mipssde-6.03.00-20051020"
```

2.8 DSP Control Register

The MIPS32 DSP ASE includes a new DSP control register that has six fields: CCOND (condition code bits), OUFLAG (overflow/underflow bits), EFI (extract fail indicator bit), C (carry bit), SCOUNT (size count bits), and POS (position bits). The compiler treats the SCOUNT and POS fields as global variables, such that instructions that modify SCOUNT or POS are never optimized away. These instructions include WRDSP, EXTPDP, EXTPDPV, and MTHLIP. A function call that jumps to a function containing WRDSP, EXTPDP, EXTPDPV, or MTHLIP is also never deleted by the compiler.

For correctness, programmers must assume that a function call clobbers all fields of the DSP control register. That is, programmers cannot depend on the values in CCOND, OUFLAG, EFI or C across a function-call boundary. They must re-initialize the values of CCOND, OUFLAG, EFI or C before using them. Note that because SCOUNT and POS fields are treated as global variables, the values of SCOUNT and POS are always valid across function-call boundaries and can be used without re-initialization.

The following example shows possibly incorrect code. The first intrinsic "__builtin_mips_addsc" sets the carry bit (C) in the DSP control register, and the second intrinsic "__builtin_mips_addwc" reads the carry bit (C) from the DSP control register. However, a function call "func" inserted between "__builtin_mips_addsc" and "__builtin_mips_addwc" may change the carry bit to affect the correct result of "__builtin_mips_addwc".

```
Ex:
int test6 (int a, int b, int c, int d)
{
  __builtin_mips_addsc (a, b);
  func();
  return __builtin_mips_addwc (c, d);
}
```

The previous example may be corrected by moving "func" before the first intrinsic or after the second intrinsic as follows.

```
Ex:
int test7 (int a, int b, int c, int d)
{
  func();
  __builtin_mips_addsc (a, b);
  return __builtin_mips_addwc (c, d);
}
/* ----------------------------------------------------------- */

int test8 (int a, int b, int c, int d)
{
  int i;
  __builtin_mips_addsc (a, b);
  i = __builtin_mips_addwc (c, d);
  func();
  return i;
}
```

3 Introduction to C-Based Intrinsics for the MIPS32® DSP ASE

This section provides a basic introduction to all the intrinsics supported for the MIPS32 DSP ASE. (Note that if an intrinsic is for the DSP ASE Revision 2, we add "DSPR2" in the subsection title.) The intrinsics are illustrated using examples, and some usage tips are provided as well. They are categorized by function and data size type as follows:

- Intrinsics to access and use the DSP control register (Section 3.1 "Intrinsics for Instructions that Access and Use the DSP Control Register").
- Intrinsics for signed and unsigned 8-bit integers (Section 3.2 "Using Intrinsics for Signed and Unsigned 8-bit Integers").
- Intrinsics for Q15 data (Section 3.3 "Using Intrinsics for Q15 Data Type").
- Intrinsics for Q31 data (Section 3.4 "Using Intrinsics for Q31 Data Type").
- Intrinsics for mixed data types of 8-bit integers and Q15/16-bit integers (Section 3.5 "Using Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers").
- Intrinsics for mixed data types of Q15 and Q31 (Section 3.6 "Using Intrinsics for Mixed Data Types: Q15 and Q31").
- Intrinsics for 64-bit accumulators (Section 3.7 "Using Intrinsics for 64-bit Accumulators").
- Intrinsics for 32-bit integers (Section 3.8 "Using Intrinsics for 32-bit Integers").
- Intrinsics for 16-bit integers (Section 3.9 "Using Intrinsics for 16-bit Integers").
- Intrinsics for mixed data types of 16-bit integers and 32-bit integers (Section 3.10 "Using Intrinsics for Mixed Data Types: 16-bit and 32-bit Integers").

3.1 Intrinsics for Instructions that Access and Use the DSP Control Register

3.1.1 Read/Write the DSP Control Register

```
i32 __builtin_mips_rddsp (imm0_63);
void __builtin_mips_wrdsp (i32, imm0_63);
```

The immediate parameter, imm0_63 used in the two intrinsics here is a mask value used to specify which fields of the DSP control register should be read or written respectively. The correspondence of the specific bits of the mask to specific fields in the DSP control register is shown in Figure 2. As shown, bit 0 of imm0_63 is for the POS field, bit 1 of imm0_63 is for the SCOUNT field, bit 2 of imm0_63 is for the C field, bit 3 of imm0_63 is for the OUFLAG, bit 4 of imm0_63 is for the CCOND flag, and bit 5 of imm0_63 is for the EFI field. For example, to read the SCOUNT field, imm0_63 must be set to 2. To read all fields, imm0_63 must be set to 63 (1 + 2 + 4 + 8 + 16 + 32).

```
Ex:
int the_scount = (__builtin_mips_rddsp (2)) >> 7; // Read SCOUNT
int all_fields = __builtin_mips_rddsp (63); // Read all fields
```

To write the DSP control register, programmers must pass a 32-bit integer as the first parameter to "__builtin_mips_wrdsp", as well as the imm0_63 mask value that determines which fields are to be updated. The first parameter should be a 32-bit value that mimics the format of the DSP control register fields. Then, based on the mask value, the corresponding fields will be copied from this 32-bit value to the DSP control register. For example, to set the SCOUNT field to 63, (63<<7) is passed as the first parameter and second parameter imm0_63 must be 2 so that an update of the SCOUNT field is done to the value 63 from the first input. To update all bits of all fields to 1, the first parameter can be 0xFFFFFFFF with a second parameter value of 63.

```
Ex:
__builtin_mips_wrdsp (63<<7, 2); // Update SCOUNT to 63
__builtin_mips_wrdsp (0xFFFFFFFF, 63); // Update all bits of fields to 1
```

Figure 2 Mask Value to Access the MIPS32 DSP Control Register

Bit	31–28	27–24	23–16	15	14	13	12–7	6	5–0
	0	CCOND	OUFLAG	0	EFI	C	SCOUNT	0	POS
IMM0_63 =>		16	8		32	4	2		1

3.1.2 Branch on Greater Than or Equal to 32 in POS

```
i32 __builtin_mips_bposge32 ();
```

This intrinsic returns 1 if the value of the POS field is greater than or equal to 32. Otherwise, the intrinsic returns 0. Programmers can use "__builtin_mips_bposge32" inside a condition test, and the compiler will optimize the code to generate the "bposge32" instruction as follows.

```
Ex:
void test9 ()
{
if (__builtin_mips_bposge32())
  result_is_true();
else
  result_is_false();
}
# Generated Assembly
test9:
        .set    noreorder
        .set    nomacro
        bposge32        .L3
        nop
        j       result_is_false
        nop
        .align  3
.L3:
        j       result_is_true
        nop
```

3.2 Using Intrinsics for Signed and Unsigned 8-bit Integers

This section introduces intrinsics that operate on signed and unsigned 8-bit integers in register SIMD fashion by using a "v4i8" data type. If the programmer wants to perform an operation such as add on a single 8-bit item, then these

intrinsics can still be used by ignoring the other three un-used elements inside the “v4i8” variable. Each set of intrinsics for operations are listed below, with simple examples of their usage.

3.2.1 Unsigned Add/Subtract with Optional Saturation

```
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 0xFF};
v4i8 b = {2, 4, 6, 8};
v4i8 r1, r2, r3, r4;
r1 = __builtin_mips_addu_qb (a, b);   // r1 will be {3, 6, 9, 7}
r2 = __builtin_mips_addu_s_qb (a, b); // r2 will be {3, 6, 9, 0xFF}
r3 = __builtin_mips_subu_qb (a, b);   // r3 will be {0xFF, 0xFE, 0xFD, 0xF7}
r4 = __builtin_mips_subu_s_qb (a, b); // r4 will be {0, 0, 0, 0xF7}
```

3.2.2 Unsigned Reduction Add

```
i32 __builtin_mips_raddu_w_qb (v4i8);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
int sum = __builtin_mips_raddu_w_qb (a); // sum will be 1 + 2 + 3 + 4 = 10
```

3.2.3 Shift Left/Right Logical

```
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
v4i8 b = {128, 64, 32, 16};
v4i8 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shll_qb (a, 1); // r1 will be {2, 4, 6, 8}
r2 = __builtin_mips_shll_qb (a, shift_amount); // r2 will be {4, 8, 12, 16}
r3 = __builtin_mips_shrl_qb (b, 3); // r3 will be {16, 8, 4, 2}
r4 = __builtin_mips_shrl_qb (b, shift_amount); // r4 will be {32, 16, 8, 4}
```

3.2.4 Dot Product with Accumulate/Subtract

```
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);
```

NOTES:
1. The result will be a 64-bit integer.
2. The processor endianness of the data affects the format of the result.
3. Using the same “a64” variable for both the target and the first parameter could result in better performance.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {1, 2, 3, 4};
v4i8 b = {4, 5, 6, 7};
a64 ac1, ac2, ac3, ac4;
ac1 = ac2 = ac3 = ac4 = 0;
ac1 = __builtin_mips_dpau_h_qbl (ac1, a, b); // ac1 will be 0 + 1*4 + 2*5 = 14
ac2 = __builtin_mips_dpau_h_qbr (ac2, a, b); // ac2 will be 0 + 3*6 + 4*7 = 46
ac3 = __builtin_mips_dpsu_h_qbl (ac3, a, b); // ac3 will be 0 - (1*4 + 2*5) = -14
ac4 = __builtin_mips_dpsu_h_qbr (ac4, a, b); // ac4 will be 0 - (3*6 + 4*7) = -46
```

3.2.5 Replicate a Fixed Byte Value into SIMD Elements

```
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (i32);
#--------------------------------------------------------------------------
Ex:
v4i8 a, b;
int value = 100;
a = __builtin_mips_repl_qb (10); // a will be {10, 10, 10, 10}
b = __builtin_mips_repl_qb (value); // b will be {100, 100, 100, 100}
```

3.2.6 Compare Unsigned (DSPR1/DSPR2)

```
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);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8); // DSPR2
```

Note that the first three intrinsics update the condition code bits of the DSP control register, but the middle three intrinsics write the condition code bits to a specified general purpose register. The last three intrinsics do the both.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {1, 4, 10, 8};
v4i8 b = {1, 2, 100, 8};
int r1, r2, r3;
__builtin_mips_cmpu_eq_qb (a, b); // CCOND bits will be 9 (= 1001b)
__builtin_mips_cmpu_lt_qb (a, b); // CCOND bits will be 2 (= 0010b)
__builtin_mips_cmpu_le_qb (a, b); // CCOND bits will be 11 (= 1011b)
r1 = __builtin_mips_cmpgu_eq_qb (a, b); // r1 will be 9
r2 = __builtin_mips_cmpgu_lt_qb (a, b); // r2 will be 2
r3 = __builtin_mips_cmpgu_le_qb (a, b); // r3 will be 11
r1 = __builtin_mips_cmpgdu_eq_qb (a, b); // Both CCOND bits and r1 will be 9
r2 = __builtin_mips_cmpgdu_lt_qb (a, b); // Both CCOND bits and r2 will be 2
r3 = __builtin_mips_cmpgdu_le_qb (a, b); // Both CCOND bits and r3 will be 11
```

3.2.7 Pick Based on Condition Code Bits

```
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
```

Note that this intrinsic is usually used together with the first three compare intrinsics in Section 3.2.6.

```
#--------------------------------------------------------------------------
Ex:
v4i8 a = {1, 4, 10, 8};
v4i8 b = {1, 2, 100, 8};
v4i8 r;
__builtin_mips_cmpu_eq_qb (a, b); // CCOND bits will be 9 (= 1001b)
r = __builtin_mips_pick_qb (a, b); // r will be {1, 2, 100, 8}
```

3.2.8 Find Absolute Value (DSPR2)

```
v4i8 __builtin_mips_absq_s_qb (v4i8); // DSPR2
#--------------------------------------------------------------------------
Ex:
v4i8 a = {-1, -128, 1, 127};
v4i8 r;
r = __builtin_mips_absq_s_qb (a); // r will be {1, 127, 1, 127}
/* Note that the absolute value of -128 is 128 that is represented by the maximum
value as 127. */
```

3.2.9 Unsigned Add and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8); // DSPR2
#--------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
v4i8 b = {0x80, 0x80, 0x80, 0x80};
v4i8 r1, r2;
r1 = __builtin_mips_adduh_qb (a, b); // r1 will be {0x40, 0x41, 0x41, 0x42}
r2 = __builtin_mips_adduh_r_qb (a, b); // r2 will be {0x41, 0x41, 0x44, 0x42}
```

3.2.10 Shift Right Arithmetic with Optional Rounding (DSPR2)

```
  v4i8 __builtin_mips_shra_qb (v4i8, imm0_7); // DSPR2
  v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7); // DSPR2
  v4i8 __builtin_mips_shra_qb (v4i8, i32); // DSPR2
  v4i8 __builtin_mips_shra_r_qb (v4i8, i32); // DSPR2
#--------------------------------------------------------------------------
Ex:
v4i8 a = {0x40, 0x20, 0x10, 0x0F};
v4i8 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shra_qb (a, 2); // r1 will be {0x10, 0x08, 0x04, 0x3}
r2 = __builtin_mips_shra_r_qb (a, 2); // r2 will be {0x10, 0x08, 0x04, 0x4}
r3 = __builtin_mips_shra_qb (a, shift_amount); // r3 will be {0x10, 0x08, 0x04, 0x3}
r4 = __builtin_mips_shra_r_qb (a, shift_amount); // r4 will be {0x10, 0x08, 0x04, 0x4}
```

3.2.11 Unsigned Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8); // DSPR2
#--------------------------------------------------------------------------
```

```
Ex:
v4i8 a = {0x80, 0x80, 0x80, 0x80};
v4i8 b = {1, 2, 3, 4};
v4i8 r1, r2;
r1 = __builtin_mips_subuh_qb (a, b); // r1 will be {0x3F, 0x3F, 0x3E, 0x3E}
r2 = __builtin_mips_subuh_r_qb (a, b); // r2 will be {0x40, 0x3F, 0x3F, 0x3E}
```

3.3 Using Intrinsics for Q15 Data Type

This section introduces intrinsics that operate on Q15 data present in register SIMD fashion by using a “v2q15” data type. If the programmer wants to perform the specified operation on a single data in the register, then these intrinsics can still be used while ignoring the other element inside the “v2q15” variable.

3.3.1 Add/Subtract with Optional Saturation

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0000, 0x8000};
v2q15 b = {0x8000, 0x8000};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_addq_ph (a, b); // r1 will be {0x8000, 0x0000}
r2 = __builtin_mips_addq_s_ph (a, b); // r2 will be {0x8000, 0x8000}
r3 = __builtin_mips_subq_ph (a, b); // r3 will be {0x8000, 0x0000}
r4 = __builtin_mips_subq_s_ph (a, b); // r4 will be {0x7FFF, 0x0000}
```

3.3.2 Find Absolute Value

```
v2q15 __builtin_mips_absq_s_ph (v2q15);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0xFFFF, 0x8000};
v2q15 r;
r = __builtin_mips_absq_s_ph (a); // r will be {0x0001, 0x7FFF}
/* Note that the value of 0x8000 is -1 in Q15.  The absolute value of -1 is 1 that
is represented by the maximum value as 0x7FFF in Q15.  */
```

3.3.3 Shift Left Logical with Optional Saturation

```
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);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0001, 0x8000};
v2q15 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shll_ph (a, 1); // r1 will be {0x0002, 0x0000}
r2 = __builtin_mips_shll_ph (a, shift_amount); // r2 will be {0x0004, 0x0000}
```

```
r3 = __builtin_mips_shll_s_ph (a, 1); // r3 will be {0x0002, 0x8000}
r4 = __builtin_mips_shll_s_ph (a, shift_amount); // r4 will be {0x0004, 0x8000}
```

3.3.4 Shift Right Arithmetic with Optional Rounding

```
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);
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x7FFF, 0x8000};
v2q15 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shra_ph (a, 1); // r1 will be {0x3FFF, 0xC000}
r2 = __builtin_mips_shra_ph (a, shift_amount); // r2 will be {0x1FFF, 0xE000}
r3 = __builtin_mips_shra_r_ph (a, 1); // r3 will be {0x4000, 0xC000}
r4 = __builtin_mips_shra_r_ph (a, shift_amount); // r4 will be {0x2000, 0xE000}
```

3.3.5 Multiply with Rounding and Saturation (Q15 x Q15 => Q15)

```
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x7FFF, 0x8000};
v2q15 b = {0x7FFF, 0x8000};
v2q15 r;
r = __builtin_mips_mulq_rs_ph (a, b); // r will be {0x7FFE, 0x7FFF}
```

3.3.6 Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31)

```
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
```

NOTES:
1. The result will be in the Q32.31 format.
2. Using the same "a64" variable for the target and the first parameter could lead to better performance.

```
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpaq_s_w_ph (ac1, a, b); // ac1 will be 0 + (1*2)<<1 +
                                              // 0x7FFFFFFF = 0x0000000080000003
ac2 = __builtin_mips_dpsq_s_w_ph (ac2, a, b); // ac2 will be 0 - (1*2)<<1 -
                                              // 0x7FFFFFFF = 0xFFFFFFFF7FFFFFFD
```

3.3.7 Multiply and Subtract and Accumulate (Q15 x Q15 => Q32.31)

```
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);
```

NOTES:
1. The result will be in the Q32.31 format.

2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#------------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1 = 0;
ac1 = __builtin_mips_mulsaq_s_w_ph (ac1, a, b); // ac1 will be 0 + (1*2)<<1 -
                                                // 0x7FFFFFFF = 0xFFFFFFFF80000005
```

3.3.8 Multiply with Accumulate a Single Element (Q15 x Q15 => Q31)

```
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);
```

NOTES:
1. The result will be in the Q31 format.
2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#------------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1, ac2, ac3, ac4;
ac1 = ac2 = 0;
ac3 = ac4 = 0x7FFFFFF0;
ac1 = __builtin_mips_maq_s_w_phl (ac1, a, b); // ac1 will be 0 + (1*2)<<1 =
                                              // 0x4
ac2 = __builtin_mips_maq_s_w_phr (ac2, a, b); // ac2 will be 0 + 0x7FFFFFFF =
                                              // 0x7FFFFFFF
ac3 = __builtin_mips_maq_sa_w_phl (ac3, a, b); // ac3 will be 0x7FFFFFF0 +
                                               // (1*2)<<1 = 0x7FFFFFF4
ac4 = __builtin_mips_maq_sa_w_phr (ac4, a, b); // ac4 will be 0x7FFFFFF0 +
                                               // 0x7FFFFFFF = 0x7FFFFFFF
```

3.3.9 Multiply Vector Fractional Left/Right Half-Words to Expanded Width Product with Saturation (Q15 x Q15 => Q31)

```
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
```

NOTES:
1. The result will be in the Q31 format.
2. The processor endianness affects the format of the result.

```
#------------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x8000};
v2q15 a = {0x5678, 0x8000};
q31 r1, r2;
r1 = __builtin_mips_muleq_s_w_phl (a, b); // r1 will be 0x0C4C00C0
r2 = __builtin_mips_muleq_s_w_phr (a, b); // r2 will be 0x7FFFFFFF
```

3.3.10 Replicate a Fixed Half-word into Elements

```
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
```

Note that for the immediate version, imm_n512_511 will be sign-extended to a 16-bit value and replicated into each SIMD element.

```
#--------------------------------------------------------------------------
Ex:
v2q15 r1, r2;
int value = 0x1234;
r1 = __builtin_mips_repl_ph (-512); // r1 will be {0xFE00, 0xFE00};
r2 = __builtin_mips_repl_ph (value); // r2 will be {0x1234, 0x1234};
```

3.3.11 Compare

```
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
Ex:
v2q15 a = {0x1111, 0x1234};
v2q15 b = {0x4444, 0x1234};
__builtin_mips_cmp_eq_ph (a, b); // CCOND bits will be 1 (= 01b)
__builtin_mips_cmp_lt_ph (a, b); // CCOND bits will be 2 (= 10b)
__builtin_mips_cmp_le_ph (a, b); // CCOND bits will be 3 (= 11b)
```

3.3.12 Pick Based on Condition Code Bits

```
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
```

Note that this intrinsic is usually used together with the compare intrinsics in Section 3.3.11.

```
#--------------------------------------------------------------------------
Ex:
v2q15 a = {0x1111, 0x1234};
v2q15 b = {0x4444, 0x1234};
v2q15 r;
__builtin_mips_cmp_eq_ph (a, b); // CCOND bits will be 1 (= 01b)
r = __builtin_mips_pick_ph (a, b); // r will be {0x4444, 0x1234}
```

3.3.13 Pack from the Right and Left

```
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
```

Note that the endianness affects the result.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1111, 0x2222};
v2q15 b = {0x3333, 0x4444};
v2q15 r;
r = __builtin_mips_packrl_ph (a, b); // r will be {0x2222, 0x3333}
```

3.3.14 Multiply with Saturation (Q15 x Q15 => Q15) (DSPR2)

```
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15); // DSPR2
```

```
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x7FFF, 0x8000};
v2q15 b = {0x7FFF, 0x8000};
v2q15 r;
r = __builtin_mips_mulq_s_ph (a, b); // r will be {0x7FFE, 0x7FFF}
```

3.3.15 Add and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15); // DSPR2
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x1000, 0x1000};
v2q15 b = {0x1001, 0x1000};
v2q15 r1, r2;
r1 = __builtin_mips_addqh_ph (a, b); // r1 will be {0x1000, 0x1000}
r2 = __builtin_mips_addqh_r_ph (a, b); // r1 will be {0x1001, 0x1000}
```

3.3.16 Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15); // DSPR2
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x1000, 0x1000};
v2q15 b = {0x1001, 0x1000};
v2q15 r1, r2;
r1 = __builtin_mips_subqh_ph (a, b); // r1 will be {0xFFFF, 0x0000}
r2 = __builtin_mips_subqh_r_ph (a, b); // r1 will be {0x0000, 0x0000}
```

3.3.17 Cross Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31) (DSPR2)

```
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
```

NOTES:
1. The result will be in the Q32.31 format.
2. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x0002, 0x8000};
v2q15 b = {0x8000, 0x0003};
a64 ac1, ac2, ac3, ac4;
ac1 = ac2 = ac3 = ac4 = 0;
ac1 = __builtin_mips_dpaqx_s_w_ph (ac1, a, b);  // ac1 will be
                                               //  0 + (2*3)<<1 + 0x7FFFFFFF
                                                    // = 0x000000008000000B
ac2 = __builtin_mips_dpaqx_sa_w_ph (ac2, a, b); // ac2 will be saturated to
                                                // 0x000000007FFFFFFF
ac3 = __builtin_mips_dpsqx_s_w_ph (ac3, a, b); // ac3 will be
                                              // 0 - (2*3)<<1 - 0x7FFFFFFF
                                              // = 0xFFFFFFFF7FFFFFF5
ac4 = __builtin_mips_dpsqx_sa_w_ph (ac4, a, b); // ac4 will be saturated
```

```
                                                  // to 0xFFFFFFFF80000000
```

3.4 Using Intrinsics for Q31 Data Type

3.4.1 Add/Subtract with Saturation

```
q31 __builtin_mips_addq_s_w (q31, q31);
q31 __builtin_mips_subq_s_w (q31, q31);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x12345678;
q31 b = 0x7FFFFFFF;
q31 r1, r2;
r1 = __builtin_mips_addq_s_w (a, b); // r1 will be 0x7FFFFFFF
r2 = __builtin_mips_subq_s_w (a, b); // r2 will be 0x92345679
```

3.4.2 Find Absolute Value

```
q31 __builtin_mips_absq_s_w (q31);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 r;
r = __builtin_mips_absq_s_w (a); // r will be 0x7FFFFFFF
/* Note that the value of 0x80000000 is -1 in Q31.  The absolute value of -1 is 1
that is represented by the maximum value as 0x7FFFFFFF in Q31.  */
```

3.4.3 Shift Left Logical with Saturation

```
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x70000000;
q31 r1, r2;
int shift_amount = 2;
r1 = __builtin_mips_shll_s_w (a, 1); // r1 will be 0x7FFFFFFF
r2 = __builtin_mips_shll_s_w (a, shift_amount); // r2 will be 0x7FFFFFFF
```

3.4.4 Shift Right Arithmetic with Rounding

```
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x7FFFFFFF;
q31 r1, r2;
int shift_amount = 2;
r1 = __builtin_mips_shra_r_w (a, 1); // r1 will be 0x40000000
r2 = __builtin_mips_shra_r_w (a, shift_amount); // r2 will be 0x20000000
```

3.4.5 Dot Product with Accumulate/Subtract (Q31 x Q31 => Q63)

```
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
```

NOTES:
1.The result will be in the Q63 format.
2. Using the same "a64" variable for the target and the first parameter could lead to better performance.

```
#--------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 b = 0x80000000;
a64 ac1, ac2;
ac1 = ac2 = 1;
ac1 = __builtin_mips_dpaq_sa_l_w (ac1, a, b); // ac1 will be 0x7FFFFFFFFFFFFFFF
ac2 = __builtin_mips_dpaq_sa_l_w (ac2, a, b); // ac2 will be 0x8000000000000002
```

3.4.6 Multiply with Rounding and Saturation (Q31 x Q31 => Q31) (DSPR2)

```
q31 __builtin_mips_mulq_rs_w (q31, q31); // DSPR2
#--------------------------------------------------------------------------
Ex:
q31 a = 0x7FFFFFFF;
q31 b = 0x00000001;
q31 r;
r = __builtin_mips_mulq_rs_w (a, b); // r will be 0x00000001
```

3.4.7 Multiply with Saturation (Q31 x Q31 => Q31) (DSPR2)

```
q31 __builtin_mips_mulq_s_w (q31, q31); // DSPR2
#--------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 b = 0x00000001;
q31 r;
r = __builtin_mips_mulq_s_w (a, b); // r will be 0x00000000
```

3.4.8 Add and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
q31 __builtin_mips_addqh_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_r_w (q31, q31); // DSPR2
#--------------------------------------------------------------------------
Ex:
q31 a = 0x10000000;
q31 b = 0x10000001;
q31 r1, r2;
r1 = __builtin_mips_addqh_w (a, b); // r1 will be 0x10000000
r2 = __builtin_mips_addqh_r_w (a, b); // r2 will be 0x10000001
```

3.4.9 Subtract and Right Shift to Halve Results with Optional Rounding (DSPR2)

```
q31 __builtin_mips_subqh_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_r_w (q31, q31); // DSPR2
```

```
#----------------------------------------------------------------------------
Ex:
q31 a = 0x10000000;
q31 b = 0x10000001;
q31 r1, r2;
r1 = __builtin_mips_subqh_w (a, b); // r1 will be 0xFFFFFFFF
r2 = __builtin_mips_subqh_r_w (a, b); // r2 will be 0x00000000
```

3.5 Using Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers

3.5.1 Precision Reduce Four Fractional Half-words to Four Bytes

```
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x5678};
v2q15 b = {0x1111, 0x2222};
v4i8 r;
r = __builtin_mips_precrq_qb_ph (a, b); // r will be {0x12, 0x56, 0x11, 0x22}
```

3.5.2 Precision Reduce Unsigned Four Fractional Half-words to Four Bytes with Saturation

```
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x7F79, 0xFFFF};
v2q15 b = {0x7F81, 0x2000};
v4i8 r;
r = __builtin_mips_precrqu_s_qb_ph (a, b); // r will be {0xFE, 0x00, 0xFF, 0x40}
```

3.5.3 Precision Expand Two Unsigned Bytes to Fractional Half-word Values

```
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);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x12, 0x34, 0x56, 0x78};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_precequ_ph_qbl (a, b); // r1 will be {0x0900, 0x1A00}
r2 = __builtin_mips_precequ_ph_qbr (a, b); // r2 will be {0x2B00, 0x3C00}
r3 = __builtin_mips_precequ_ph_qbla (a, b); // r3 will be {0x0900, 0x2B00}
r4 = __builtin_mips_precequ_ph_qbra (a, b); // r4 will be {0x1A00, 0x3C00}
```

3.5.4 Precision Expand Two Unsigned Bytes to Unsigned Integer Half-words

```
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);
```

Note that the processor endianness affects the format of the result.

```
#---------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x12, 0x34, 0x56, 0x78};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_preceu_ph_qbl (a, b); // r1 will be {0x0012, 0x0034}
r2 = __builtin_mips_preceu_ph_qbr (a, b); // r2 will be {0x0056, 0x0078}
r3 = __builtin_mips_preceu_ph_qbla (a, b); // r3 will be {0x0012, 0x0056}
r4 = __builtin_mips_preceu_ph_qbra (a, b); // r4 will be {0x0034, 0x0078}
```

3.5.5 Multiply Unsigned Vector Left/Right Bytes with Half-Words to Half Word Products with Saturation (Int8 x Q15 => Q15)

```
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#---------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x1, 0x3, 0x5, 0x7};
v2q15 b = {0x1234, 0x5678};
v2q15 r1, r2;
r1 = __builtin_mips_muleu_s_ph_qbl (a, b); // r1 will be {0x1234, 0xFFFF}
r2 = __builtin_mips_muleu_s_ph_qbr (a, b); // r2 will be {0x5B04, 0xFFFF}
```

3.5.6 Precision Reduce Four Integer Half-words to Four Bytes (DSPR2)

```
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16); // DSPR2
```

Note that the processor endianness affects the format of the result.

```
#---------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2i16 a = {0x7F79, 0xFFFF};
v2i16 b = {0x7F81, 0x2000};
v4i8 r;
r = __builtin_mips_precr_qb_ph (a, b); // r will be {0x79, 0xFF, 0x81, 0x00}
```

3.6 Using Intrinsics for Mixed Data Types: Q15 and Q31

3.6.1 Precision Reduce Two Fractional Words to Two Half-Words

```
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
#---------------------------------------------------------------------------
Ex:
q31 a = {0x12345678};
q31 b = {0x11112222};
```

```
v2q15 r;
r = __builtin_mips_precrq_ph_w (a, b); // r will be {0x1234, 0x1111}
```

3.6.2 Precision Reduce Two Fractional Words to Two Half-Words with Rounding and Saturation

```
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
#---------------------------------------------------------------------------
Ex:
q31 a = {0x7000FFFF};
q31 b = {0x80000000};
v2q15 r;
r = __builtin_mips_precrq_rs_ph_w (a, b); // r will be {0x7001, 0x8000}
```

3.6.3 Precision Expand a Fractional Half-word to a Fractional Word Value

```
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
```

Note that the endianness affects the result.

```
#---------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x5678};
q31 r1, r2;
r1 = __builtin_mips_preceq_w_phl (a, b); // r1 will be 0x12340000
r2 = __builtin_mips_preceq_w_phr (a, b); // r2 will be 0x56780000
```

3.7 Using Intrinsics for 64-bit Accumulators

3.7.1 Extract a Value with Right Shift

```
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_r_w (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, imm0_31);
i32 __builtin_mips_extr_rs_w (a64, i32);
#---------------------------------------------------------------------------
Ex:
a64 ac1 = 0x8123456712345678;
i32 shift_amount = 31;
i32 r1, r2, r3, r4, r5, r6;
r1 = __builtin_mips_extr_w (ac1, 1); // r1 will be 0x891A2B3C
r2 = __builtin_mips_extr_w (ac1, shift_amount); // r2 will be 0x02468ACE
r3 = __builtin_mips_extr_r_w (ac1, 4); // r3 will be 0x71234568
r4 = __builtin_mips_extr_r_w (ac1, shift_amount); // r4 will be 0x02468ACE
r5 = __builtin_mips_extr_rs_w (ac1, 4); // r5 will be 0x80000000
r6 = __builtin_mips_extr_rs_w (ac1, shift_amount); // r6 will be 0x80000000
```

3.7.2 Extract Half-word with Right Shift and Saturate

```
i32 __builtin_mips_extr_s_h (a64, imm0_31);
i32 __builtin_mips_extr_s_h (a64, i32);
```

Note that the 16-bit result is sign-extended to a 32-bit result.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0xFFFFF81230000000;
i32 shift_amount = 4;
i32 r1, r2;
r1 = __builtin_mips_extr_s_h (ac1, 28); // r1 will be 0xFFFF8123
r2 = __builtin_mips_extr_s_h (ac1, shift_amount); // r2 will be 0xFFFF8000
```

3.7.3 Extract Bit from an Arbitrary Position

```
i32 __builtin_mips_extp (a64, imm0_31);
i32 __builtin_mips_extp (a64, i32);
```

Note that the “imm0_31” + 1 bits between “POS” and “POS” - “imm0_31” are extracted and zero-extended to a 32-bit result. So, if X bits are extracted, X-1 should be used as the second parameter. The POS field can be set by using “__builtin_mips_wrdsp”.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
i32 r1, r2;
int the_size = 3;
__builtin_mips_wrdsp (35, 1); // Write 35 to the POS field
r1 = __builtin_mips_extp (ac1, 31); // r1 will be 0x88765432
r2 = __builtin_mips_extp (ac1, the_size); // r2 will be 0x8
```

3.7.4 Extract Bit from an Arbitrary Position and Decrement POS

```
i32 __builtin_mips_extpdp (a64, imm0_31);
i32 __builtin_mips_extpdp (a64, i32);
```

Note that this intrinsic is the same as the ones in Section 3.7.3, except that in addition, the POS field is decremented by the number of extracted bits, “imm0_31” + 1 or “i32” + 1.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x123456789ABCDEF0;
i32 r1, r2;
int the_size = 7;
__builtin_mips_wrdsp (35, 1); // Write 35 to the POS field
r1 = __builtin_mips_extpdp (ac1, 3); // r1 will be 0x8, and POS will be 31
r2 = __builtin_mips_extpdp (ac1, the_size); // r2 will be 0x9a, and POS will be 23
```

3.7.5 Shift an Accumulator Value

```
a64 __builtin_mips_shilo (a64, imm_n32_31);
a64 __builtin_mips_shilo (a64, i32);
```

Note that using the same a64 variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
int shift_amount = -8;
ac1 = __builtin_mips_shilo (ac1, 8); // ac1 will be 0x0012345678876543
ac1 = __builtin_mips_shilo (ac1, shift_amount); // ac1 will be 0x1234567887654300
```

3.7.6 Copy the LO to HI and a Value to LO and Increment POS by 32

```
a64 __builtin_mips_mthlip (a64, i32);
```

Note that using the same a64 variable for the target and the first parameter could lead to better performance.

```
#---------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
int b = 0x11112222
__builtin_mips_wrdsp (0, 1); // Write 0 to the POS field
ac1 = __builtin_mips_mthlip (ac1, b); // ac1 will be 0x8765432111112222,
                                      // and POS will be 32
```

3.8 Using Intrinsics for 32-bit Integers

3.8.1 Add and Set Carry/Add with Carry

```
i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);
```

Note that these two intrinsics can be used to add two 64-bit operands, each spread across two GPRs. The lower 32 bits are calculated first, then the carry from this addition is fed to the add of the upper 32 bit values.

```
#---------------------------------------------------------------------------
Ex:
int i1 = 0;
int i2 = 0xFFFFFFFF;
int j1 = 1;
int j2 = 1;
int r1, r2;
r2 = __builtin_mips_addsc (i2, j2); // r2 will be 0xFFFFFFFF+1 = 0 and C will be 1
r1 = __builtin_mips_addwc (i1, j1); // r1 will be 0 + 1 + 1(C) = 2
```

3.8.2 Modular Subtraction on an Index Value

```
i32 __builtin_mips_modsub (i32, i32);
```

Note that this intrinsic can be used to implement a circular buffer. The first parameter is the current index, that will be checked against zero. If the index is zero, the new index will be rolled back to the top of the buffer, assigned from the bit 23 to 8 of the second parameter. If the index is not zero, the new index will be decremented by the size of the element, assigned from the bit 7 to 0 of the second parameter.

```
#---------------------------------------------------------------------------
Ex:
int index = 20;
int element = 0x1402;
while (1)
{
  index = __builtin_mips_modsub (index, element);
}
/* ‘index’ will be 20, 18, 16, ..., 4, 2, 0, 20, 18, 16, ... */
```

3.8.3 Bit Reverse a Half-word

```
i32 __builtin_mips_bitrev (i32);
```

```
#---------------------------------------------------------------------------
Ex:
int a = 0x1234; // 0001 0010 0011 0100
int r;
r = __builtin_mips_bitrev (a); // r will be 0x2c48 (0010 1100 0100 1000)
```

3.8.4 Insert Bit Field Variable

```
i32 __builtin_mips_insv (i32, i32);
```

Note that using the same variable for the target and the first parameter could lead to better performance. This intrinsic inserts the value of the second parameter to the first parameter. The size to be extracted from the second parameter is specified in the SCOUNT field. The position to be inserted in the first parameter is specified in the POS field.

```
#---------------------------------------------------------------------------
Ex:
int a = 0x12345678;
int r = 0xFFFFFFFF;
__builtin_mips_wrdsp ((16<<7)+4, 3); // set SCOUNT to 16, and set POS to 4
r = __builtin_mips_insv (r, a); // The lowest 16-bit value of a is inserted to r
                                // at bit 4. r will be 0xFFF5678F.
```

3.8.5 Load Unsigned Byte/Halfword/Word Indexed

```
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
```

NOTES:
1, The first parameter is the base of the array, and the second parameter is the offset in byte.
2.The returned value is zero-extended for "__builtin_mips_lbux", and sign-extended for "__builtin_mips_lhx" and "__builtin_mips_lwx" to 32-bit integers.

```
#---------------------------------------------------------------------------
Ex:
char array_a[100];
short array_b[100];
int array_c[100];
int offset = 20;
int r1, r2, r3;
r1 = __builtin_mips_lbux ((void *)array_a, offset);
r2 = __builtin_mips_lhx ((void *)array_b, offset);
r3 = __builtin_mips_lwx ((void *)array_c, offset);
```

3.8.6 Signed Multiply and Add

```
a64 __builtin_mips_madd (a64, i32, i32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_madd (a, b, c); // a will be 1 + 2 * (-3) = -5
```

3.8.7 Unsigned Mulitply and Add

```
a64 __builtin_mips_maddu (a64, ui32, ui32);
#---------------------------------------------------------------------------
a64 a = 1;
ui32 b = 2;
ui32 c = 3;
a = __builtin_mips_maddu (a, b, c); // a will be 1 + 2 * 3 = 7
```

3.8.8 Signed Multiply and Subtract

```
a64 __builtin_mips_msub (a64, i32, i32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_msub (a, b, c); // a will be 1 - 2 * (-3) = 7
```

3.8.9 Unsigned Multiply and Subtract

```
a64 __builtin_mips_msubu (a64, ui32, ui32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
ui32 b = 2;
ui32 c = 3;
a = __builtin_mips_msubu (a, b, c); // a will be 1 - 2 * 3 = -5
```

3.8.10 Signed Multiply

```
a64 __builtin_mips_mult (i32, i32);
#---------------------------------------------------------------------------
a64 a;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_mullt (b, c); // a will be 2 * (-3) = -6
```

3.8.11 Unsigned Multiply

```
a64 __builtin_mips_multu (ui32, ui32);
#---------------------------------------------------------------------------
Ex:
a64 a;
u32 b = 2;
u32 c = 3;
a = __builtin_mips_mulltu (b, c); // a will be 2 * 3 = 6
```

3.8.12 Left Shift and Append Bits (DSPR2)

```
i32 __builtin_mips_append (i32, i32, imm0_31); // DSPR2
#---------------------------------------------------------------------------
Ex:
```

```
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_append (a, b, 4); // r will be 0x765ABCD8
```

3.8.13 Byte Align Contents from Two Registers (DSPR2)

```
i32 __builtin_mips_balign (i32, i32, imm0_3); // DSPR2
#-----------------------------------------------------------------------------
Ex:
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_balign (a, b, 3); // r will be 0xCD123456
```

3.8.14 Right Shift and Prepend Bits (DSPR2)

```
i32 __builtin_mips_prepend (i32, i32, imm0_31); // DSPR2
#-----------------------------------------------------------------------------
Ex:
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_prepend (a, b, 4); // r will be 0x88765ABC
```

3.9 Using Intrinsics for 16-bit Integers

3.9.1 Unsigned Add/Subtract with Optional Saturation (DSPR2)

```
v2i16 __builtin_mips_addu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16); // DSPR2
#-----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0000, 0x8000};
v2i16 b = {0x8000, 0x8000};
v2i16 r1, r2, r3, r4;
r1 = __builtin_mips_addu_ph (a, b); // r1 will be {0x8000, 0x0000}
r2 = __builtin_mips_addu_s_ph (a, b); // r2 will be {0x8000, 0xFFFF}
r3 = __builtin_mips_subu_ph (a, b); // r3 will be {0x8000, 0x0000}
r4 = __builtin_mips_subu_s_ph (a, b); // r4 will be {0x0000, 0x0000}
```

3.9.2 Dot Product with Accumulate/Subtract (DSPR2)

```
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. Using the same “a64” variable for both the target and the first parameter could result in better performance.

```
#-----------------------------------------------------------------------------
```

```
Ex:
v2i16 a = {0x0001, 0x8000};
v2i16 b = {0x0002, 0x8000};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpa_w_ph (ac1, a, b); // ac1 will be 0 + 1*2 +
                                           // 0x40000000 = 0x0000000040000002
ac2 = __builtin_mips_dps_w_ph (ac2, a, b); // ac2 will be 0 - 1*2 -
                                           // 0x40000000 = 0xFFFFFFFFBFFFFFFE
```

3.9.3 Multiply with Optional Saturation (DSPR2)

```
v2i16 __builtin_mips_mul_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16); // DSPR2
#-----------------------------------------------------------------------------
Ex:
v2i16 a = {0x7FFF, 0x8000};
v2i16 b = {0x7FFF, 0x8000};
v2i16 r1, r2;
r1 = __builtin_mips_mul_ph (a, b); // r1 will be {0x0001, 0x0000}
r2 = __builtin_mips_mul_s_ph (a, b); // r2 will be {0x7FFF, 0x7FFF}
```

3.9.4 Multiply and Subtract and Accumulate (DSPR2)

```
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#-----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0001, 0x8000};
v2i16 b = {0x0002, 0x8000};
a64 ac1 = 0;
ac1 = __builtin_mips_mulsa_w_ph (ac1, a, b); // ac1 will be 0 + 1*2 -
                                             // 0x40000000 = 0xFFFFFFFFC0000002
```

3.9.5 Shift Right Logical (DSPR2)

```
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, i32); // DSPR2
#-----------------------------------------------------------------------------
Ex:
v2i16 a = {0x8000, 0x4000};
v2i16 r1, r2;
int shift_amount = 4;
r1 = __builtin_mips_shrl_ph (a, 4); // r1 will {0x0800, 0x0400};
r2 = __builtin_mips_shrl_ph (a, shift_amount); // r2 will {0x0800, 0x0400};
```

3.9.6 Cross Dot Product with Accumulate/Subtract (DSPR2)

```
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16); // DSPR2
```

```
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. Using the same “a64” variable for both the target and the first parameter could result in better performance.

```
#---------------------------------------------------------------------------
Ex:
v2i16 a = {0x0001, 0x0003};
v2i16 b = {0x0002, 0x0004};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpax_w_ph (ac1, a, b);  // ac1 will be
                                            //0 + 1*4 + 2*3 = 0x000000000000000A
ac2 = __builtin_mips_dpsx_w_ph (ac1, a, b); // ac2 will be
                                            // 0 - 1*4 - 2*3 = 0xFFFFFFFFFFFFFFF6
```

3.10 Using Intrinsics for Mixed Data Types: 16-bit and 32-bit Integers

3.10.1 Precision Reduce Two Integer Words to Halfwords After a Right Shift with Optional Rounding (DSPR2)

```
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31); // DSPR2
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31); // DSPR2
#---------------------------------------------------------------------------
Ex:
i32 a = 0x80000000;
i32 b = 0x7FFFFFFF;
v2i16 r1, r2;
r1 = __builtin_mips_precr_sra_ph_w (a, b, 4); // r1 will be {0x0000, 0xFFFF}
r2 = __builtin_mips_precr_sra_r_ph_w (a, b, 4); // r2 will be {0x0000, 0x0000}
```

4 Tips for Efficient Code

This section provides tips for writing efficient C programs using the MIPS32 DSP ASE. This also includes tips on toolchain usage and options that provide the most efficient code. Note that this section is very dependent on the specifics of how the GCC compiler works.

4.1 Compiler Options for Optimization and CPUs

Programmers must select proper optimization levels to compile C code to suit their purposes. For example, for maximum speed: “-O3 -funroll-loops”. For good speed with moderate code sizes: “-O2”. For minimum code sizes: “-Os”. For CodeSourcery SG++ users, please refer to “Chapter 2, Compiler Options” in *MIPS® Toolchain Specifics* (MD00624).

To let the GCC compiler schedule instructions based on the latency information, programmers must supply correct architecture and CPU options. Ex: “-mips32r2 -mtune=24ke” is recommended for MIPS32® 24KE™ cores; “-mips32r2 -mtune=34k” for MIPS32® 34K® cores.

4.2 The Use of Intrinsics versus Asm Macros

The assembler has no knowledge of the pipeline and any code written using asm macros will be treated as a single cycle latency instruction by the GCC compiler. This can lead to poor code scheduling and a lot of stalls in the resulting execution. On the other hand, the GCC compiler has knowledge of the pipeline latency of instructions and can schedule the DSP instructions correctly when programmers use intrinsics, that is "__builtin_mips_*". Hence it is important to try to avoid the use of asm macros whenever possible.

4.3 Using Accumulators

To access only HI or LO of an accumulator, programmers are recommended to use a union type as follows.

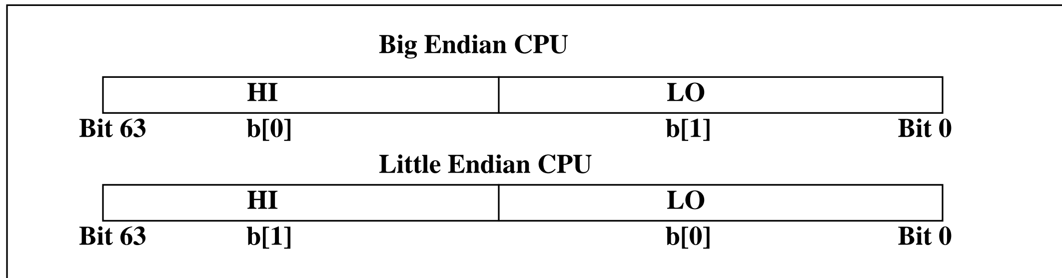
```
typedef union
{
  long long a;   // One 64-bit accumulator
  int b[2];      // 32-bit HI and LO
} a64_union;
```

Note that CPU endianness affects how to access the accumulator as shown in Figure 3. To access HI, b[0] is used in a Big Endian CPU, but b[1] is used in a Little Endian CPU. To access LO, b[1] is used in a Big Endian CPU, but b[0] is used in a Little Endian CPU.

```
Ex:
int test10 (long long a, v2q15 b, v2q15 c)
{
  a64_union temp;
  temp.a = __builtin_mips_dpaq_s_w_ph (a, b, c);
  return temp.b[0];  // Assume in a little-endian CPU we want to access LO.
}

# Generated Assembly
test10:
    mtlo   $4
    mthi   $5
    dpaq_s.w.ph  $ac0, $6, $7
    j      $31
    mflo   $2
```

Figure 3 Accessing HI and LO of Accumulators



4.4 Multiply “32-bit * 32-bit = 64-bit”

To multiply 32 bits by 32 bits to obtain a 64-bit result, we must cast the 32-bit integer to a 64-bit integer (long long) and then perform the multiplication operation as follows.

```
Ex:
long long test11 (int a, int b)
{
  return (long long) a * b;  // Same as (long long) a * (long long) b
                             // NOT the same as (long long) (a * b)
}

# Generated Assembly
test11:
    mult    $4,$5
    mflo    $2
    j       $31
    mfhi    $3
```

Combined with Section 4.3 we can multiply 32-bit by 32-bit integers and get the highest 32-bit result from HI as follows.

```
Ex:
int test12 (int a, int b)
{
  a64_union temp;
  temp.a = (long long) a * b;
  return temp.b[1];  // Assume a little-endian CPU
}

# Generated Assembly
test12:
    mult    $4,$5
    j       $31
    mfhi    $2
```

4.5 Multiply and Add “32-bit * 32-bit + 64-bit = 64-bit”

To perform multiplication and addition, we must cast the 32-bit integer to 64-bit (long long) and then perform multiplication and addition as follows.

```
Ex:
long long test13 (int a, int b, long long c)
{
  return c + (long long) a * b;
}

# Generated Assembly
test13:
    mtlo   $6
    mthi   $7
    madd   $4, $5
    mflo   $2
    j      $31
    mfhi   $3
```

4.6 Array Alignment and Data Layout

The GCC compiler provides a mechanism to specify the alignment of variables by using "__attribute__ ((aligned (*bytes*)))". The alignment is important to loading or storing SIMD variables: "v4i8" and "v2q15". If an array is aligned to a 4-byte boundary, that is, word-aligned, the GCC compiler can load or store four 8-bit data for v4i8 variables (or two 16-bit data for v2q15 variables) at a time using the load word class of instructions. The following example shows that when a char array A is aligned to a 4-byte boundary, we can cast this array to a v4i8 array and load four items to a v4i8 variable at a time by using the "lwx" instruction. However, if this char array A is not aligned to a 4-byte boundary, executing the following code will result in an address exception due to a mis-aligned load.

```
Ex: /* v4i8 Example */
char A[128] __attribute__ ((aligned (4)));
v4i8 test14 (int i)
{
  v4i8 a;
  v4i8 *myA = (v4i8 *)A;
  a = myA[i];
  return a;
}
# Generated Assembly
test14:
    lui      $2,%hi(A)
    sll      $4,$4,2
    addiu    $2,$2,%lo(A)
    j        $31
    lwx      $2,$2($4)
```

After SIMD data is loaded from memory into a register, it is best if the SIMD variables in the register are ready for use without requiring any rearrangement of the data. To avoid such data rearrangement which can reduce the benefit of parallelism, programmers must design their arrays with efficient data layout that is favorable for SIMD calculations.

4.7 GP-Relative Addressing

The GCC compiler provides an option "-G *num*" to put global or static data that is at most "*num*" bytes to the small data or bss sections. This allows using only one instruction to access data via gp-relative addressing to improve the performance. Programmers can try to increase "*num*" to include more data into small data or bss sections until these sections are full. Note that all modules should be compiled with the same "-G *num*". The following example shows how the GCC compiler accesses a 16-byte array. When compiling the example with "-G 4", calculating the base

address of the array "C" needs two instructions: "`lui $3,%hi(C)`" and "`addiu $3,$3,%lo(C)`". But, when compiling with "-G 16" to put the whole array of "C" into the small data section, only one instruction "`addiu $3,$28,%gp_rel(C)`" is required to get the base address of "C".

```
Ex:
int C[4];
void test15 (int index, int value)
{
  C[index] = value;
}

# Generated Assembly when compiling with -G 4
test15:
    lui     $3,%hi(C)
    addiu   $3,$3,%lo(C)
    sll     $4,$4,2
    addu    $4,$4,$3
    j       $31
    sw      $5,0($4)
# -----------------------------------------------------------

# Generated Assembly when compiling with -G 16
test15:
    addiu   $3,$28,%gp_rel(C)
    sll     $4,$4,2
    addu    $4,$4,$3
    j       $31
    sw      $5,0($4)
```

5 Advanced Topics and Some Complex Usage

This section provides advanced topics about changing the normal register usage, and using asm macros to generate conditional move instructions.

5.1 Fixed Registers and Register Variables

Register usage is defined by the Application Binary Interface (ABI). For example, the ABI defines that some registers are caller-saved, some are callee-saved, and a few registers are fixed (or called global) and not saved at all. When conforming to the ABI, functions are guaranteed to work with each other.

However, in very special cases where performance may be very critical, programmers may want to improve performance by avoiding the saving and restoring of registers and hence violating the ABI convention. This undertaking should be taken with caution and not normally recommended as general practice. The GCC compiler allows programmers to treat a register as fixed by using the command-line option: "-ffixed-*reg*" where *reg* must be the name of a register. When a register is fixed, the register allocation process does not touch the fixed register.

For example, the ABI defines that four accumulators ($ac0 - $ac3) are caller-saved registers, but programmers may want to dedicate one accumulator, $ac1, for a special purpose. Note that because $ac1is a 64-bit register that consists of $ac1hi and $ac1lo, "-ffixed-\$ac1hi -ffixed-\$ac1lo" is specified in the command-line options to fix HI and LO of $ac1.

The following example demonstrates that under the original ABI, the GCC compiler register allocator will allocate 64-bit variables to all accumulators. However, when $ac1 is specified to be fixed, The GCC compiler only allocates 64-bit variables to $ac0, $ac2, and $ac3.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
void test16 (a64 a[4], v2q15 b[4], v2q15 c[4])
{
  a[0] = __builtin_mips_dpaq_s_w_ph (a[0], b[0], c[0]);
  a[1] = __builtin_mips_dpaq_s_w_ph (a[1], b[1], c[1]);
  a[2] = __builtin_mips_dpaq_s_w_ph (a[2], b[2], c[2]);
  a[3] = __builtin_mips_dpaq_s_w_ph (a[3], b[3], c[3]);
}

# Generated Assembly without using "-ffixed-\$ac1hi --fixed-\$ac1lo"
# Note that $ac0, $ac1, $ac2, and $ac3 are all used.
test16:
        lw      $15,4($4)
        lw      $14,0($4)
        lw      $13,12($4)
        lw      $10,8($4)
        lw      $9,20($4)
        lw      $8,16($4)
        lw      $3,28($4)
        lw      $7,24($4)
        lw      $2,0($6)
        lw      $12,12($5)
        lw      $11,12($6)
        mtlo    $15,$ac1
        mthi    $14,$ac1
        mtlo    $13,$ac2
        lw      $25,0($5)
        lw      $15,4($5)
        lw      $24,4($6)
        lw      $13,8($5)
        lw      $14,8($6)
        mthi    $10,$ac2
        mtlo    $9,$ac3
        mthi    $8,$ac3
        mtlo    $3
        mthi    $7
        dpaq_s.w.ph     $ac1,$25,$2
        dpaq_s.w.ph     $ac2,$15,$24
        dpaq_s.w.ph     $ac3,$13,$14
        dpaq_s.w.ph     $ac0,$12,$11
        mflo    $10
        mfhi    $9
        mflo    $8,$ac1
        mfhi    $7,$ac1
        mflo    $6,$ac2
        mfhi    $5,$ac2
        mflo    $3,$ac3
        mfhi    $2,$ac3
        sw      $10,28($4)
        sw      $9,24($4)
        sw      $8,4($4)
        sw      $7,0($4)
```

```
        sw      $6,12($4)
        sw      $5,8($4)
        sw      $3,20($4)
        j       $31
        sw      $2,16($4)
# ---------------------------------------------------------------------------

# Generated Assembly when using “-ffixed-\$ac1hi --fixed-\$ac1lo”
# Note that $ac0, $ac2, and $ac3 are used.  But, $ac1 is not touched at all by the
# compiler.
test16:
        lw      $3,4($4)
        lw      $2,0($4)
        lw      $25,12($4)
        lw      $24,8($4)
        lw      $9,20($4)
        lw      $8,16($4)
        lw      $15,12($5)
        lw      $13,0($5)
        lw      $11,0($6)
        lw      $12,4($5)
        lw      $7,4($6)
        lw      $10,8($5)
        lw      $5,8($6)
        mtlo    $3,$ac2
        mthi    $2,$ac2
        lw      $3,28($4)
        lw      $2,24($4)
        mtlo    $25,$ac3
        mthi    $24,$ac3
        mtlo    $9
        mthi    $8
        dpaq_s.w.ph     $ac2,$13,$11
        lw      $14,12($6)
        dpaq_s.w.ph     $ac3,$12,$7
        dpaq_s.w.ph     $ac0,$10,$5
        mflo    $9
        mfhi    $8
        mtlo    $3
        mthi    $2
        dpaq_s.w.ph     $ac0,$15,$14
        mflo    $25
        mfhi    $24
        mflo    $6,$ac2
        mfhi    $5,$ac2
        mflo    $3,$ac3
        mfhi    $2,$ac3
        sw      $25,28($4)
        sw      $24,24($4)
        sw      $6,4($4)
        sw      $5,0($4)
        sw      $3,12($4)
        sw      $2,8($4)
        sw      $9,20($4)
        j       $31
        sw      $8,16($4)
```

To use a fixed register, programmers must associate a register variable with the explicit name of the fixed register. For example, when $ac1 is fixed, we can declare “register a64 MYACC asm (“$ac1lo”)” in a Little Endian CPU, or “register a64 MYACC asm (“$ac1hi”)” in a Big Endian CPU. Then, the global variable “MYACC” is ready to be used across all functions via directly accessing $ac1.

The following example shows that when no global register variable is used, The GCC compiler needs to load or store a 64-bit global variable from or to memory.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
a64 MYACC;
void test17 (v2q15 b, v2q15 c)
{
  MYACC = __builtin_mips_dpaq_s_w_ph (MYACC, b, c);
}

# Generated Assembly
test17:
        lw      $2,%gp_rel(MYACC)($28)
        lw      $3,%gp_rel(MYACC+4)($28)
        mtlo    $2
        mthi    $3
        dpaq_s.w.ph     $ac0,$4,$5
        mflo    $2
        mfhi    $3
        sw      $2,%gp_rel(MYACC)($28)
        j       $31
        sw      $3,%gp_rel(MYACC+4)($28)
```

However, when a register variable is used for a global variable, the overhead of storing and loading to and from memory is eliminated as follows, reducing the above 10 instructions to only 2.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
register a64 MYACC asm ("$ac1lo"); /* Assume a little-endian CPU */
void test18 (v2q15 b, v2q15 c)
{
  MYACC = __builtin_mips_dpaq_s_w_ph (MYACC, b, c);
}

# Generated Assembly by
# “sde-gcc -mips32r2 -mdsp -O4 -S -ffixed-\$ac1hi --fixed-\$ac1lo -EL 18.c”
test18:
        j       $31
        dpaq_s.w.ph     $ac1,$4,$5
```

There are a few things to note when using the technique of fixing registers for global variables.

1. When fixing accumulators, because $ac0 is the original HI and LO registers for multiplication and division instructions in MIPS32, $ac0 cannot be fixed by using “-ffixed-hi -ffixed-lo”. The rest of the accumulators, that is $ac1, $ac2, and $ac3 can be fixed.

2. When fixing $ac1, $ac2, or $ac3, programmers must ensure that no third-party or library functions that can clobber $ac1, $ac2 or $ac3 are called between accessing fixed accumulators. To practice safe programming methods,

it is probably advisable to restrict the use of fixed accumulators inside an optimized kernel that consist of only internal functions.

3. The technique of fixing registers for use as global variables can be directly applied to callee-saved registers that are $16 to $23 (s0 to s7). Programmers do not need to change s0 to s7 to be fixed registers.

5.2 Conditional Moves

Typically conditional move instructions are used instead of branch instructions to avoid the penalty from branch delay slots and mis-predicted branches. For example, the GCC compiler can generate conditional move instructions for simple C code as follows.

```
Ex:
int test19 (int true_value, int false_value, int cond)
{
if (cond)
  return true_value;
else
  return false_value;
}

# Generated Assembly
test19:
        move    $2,$4
        j       $31
        movz    $2,$5,$6
# ----------------------------------------------------------

Ex:
int test20 (int true_value, int false_value, int cond)
{
  return cond ? true_value : false_value;
}

# Generated Assembly
test20:
        move    $2,$4
        j       $31
        movz    $2,$5,$6
```

However, for complicated C code, the GCC compiler may not recognize the C patterns to generate conditional move instructions. Programmers can then use asm macros to force the GCC compiler to use conditional move instructions. The following example shows how to use an asm macro for a “movz” instruction. First, we need to assign a value “true_value” (when the condition is true) to a resultant variable “result”. Then, we pass “result”, “false_value” and “cond” to the asm macro of “movz”. Note that the asm macro uses “+d” for the output format, because the output register is also used as an input register.

```
Ex:
int test21 (int true_value, int false_value, int cond)
{
  int result = true_value;
  asm ("movz %0, %1, %2": "+d" (result): "d" (false_value), "d" (cond));
  return result;
}
```

```
# Generated Assembly
test21:
 #APP
        movz $4, $5, $6
 #NO_APP
        .set    noreorder
        .set    nomacro
        j       $31
        move    $2,$4
```

6 A Programming Example

This section presents a programming example: a 16-point FIR filter. First, we show the FIR filter in traditional C code without SIMD variables and DSP intrinsics. Then, we present the hand-tuned assembly version. Finally, the FIR filter in the efficient C code is shown.

6.1 The FIR Filter in Traditional C

The following C code implements a 16-point FIR filter without using SIMD variables and DSP intrinsics. The arrays of “coeffs” and “delay” store sixteen Q15 coefficients and sixteen Q15 delayed inputs.

```
Ex:
int i;
short x, y;
long long ac0 = 0;
for (i = 0; i < 16; i++)
{
    x = coeffs[i];
    y = delay[i];
    if ((unsigned short) x == 0x8000 && (unsigned short) y == 0x8000)
        ac0 += 0x7fffffff;
    else
        ac0 += ((x * y) << 1);
}
```

Inside a loop, a saturation condition needs to be detected when both values of “coeffs” and “delay” are 0x8000 (-1 in Q15). Moreover, to perform “Q15 x Q15” multiplication, a left shift is required after integer multiplication. This version of the FIR filter takes 536 cycles to calculate one result. (The tools used for this experiment are SDE 6.03.00-rc3 and MIPSsim 4.6.23.) The traditional C code produces inefficient binary code, so DSP programmers would write it in assembly code.

6.2 The FIR Filter in Hand-Tuned Assembly

DSP programmers pack two coefficients to a register and pack two delayed inputs to a register, so a SIMD DSP instruction “dpaq_s.w.ph” that performs saturation and “Q15 x Q15” multiplication can be applied efficiently. Instruction scheduling is performed by hand to avoid pipeline stalls. This FIR implementation can generate one result in 39 cycles which are much faster than the traditional C version in Section 6.1. The hand-tuned assembly code for the FIR filter is as follows.

```
Ex:
    mult  $0, $0
```

```
lw    $8, 0($5)
lw    $10, 0($6)
lw    $9, 4($5)
lw    $11, 4($6)
dpaq_s.w.ph $ac0, $8, $10
dpaq_s.w.ph $ac0, $9, $11
lw    $12, 8($5)
lw    $10, 8($6)
lw    $13, 12($5)
lw    $11, 12($6)
dpaq_s.w.ph $ac0, $12, $10
dpaq_s.w.ph $ac0, $13, $11
lw    $14, 16($5)
lw    $10, 16($6)
lw    $15, 20($5)
lw    $11, 20($6)
dpaq_s.w.ph $ac0, $14, $10
dpaq_s.w.ph $ac0, $15, $11
lw    $16, 24($5)
lw    $10, 24($6)
lw    $4, 28($5)
lw    $11, 28($6)
dpaq_s.w.ph $ac0, $16, $10
dpaq_s.w.ph $ac0, $4, $11
```

6.3 The FIR Filter in Efficient C

Although the hand-tuned assembly code yields good performance, it requires the programmer to manually do register allocation and code scheduling. A compromise is to write C code that uses SIMD variables and DSP intrinsics as shown.

```
Ex:
    v2q15 *my_delay = (v2q15 *)delay;
    v2q15 *my_coeffs = (v2q15 *)coeffs;
    long long ac0 = 0;
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[0], my_coeffs[0]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[1], my_coeffs[1]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[2], my_coeffs[2]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[3], my_coeffs[3]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[4], my_coeffs[4]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[5], my_coeffs[5]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[6], my_coeffs[6]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[7], my_coeffs[7]);
```

This C code does not look as clean or as readable as the traditional C version in Section 6.1, but it is efficient and calculates one result in 42 cycles which is only 7.69% slower than the hand-tuned assembly version in Section 6.2. Compared to the hand-tuned assembly code, the efficient C code has three significant advantages as follows.

1. Register allocation is done by the compiler.

2. Code scheduling is done by the compiler.

3. Load and store of SIMD data is taken care of by the compiler.

Other DSP kernels can similarly benefit from C code.

7 MIPS32 DSP Intrinsics

Note: Some parameters of intrinsics are immediate types. Programmers must pass a constant that is within the specific range in order to invoke these intrinsics. The immediate types are as follows:

```
imm0_3: the parameter must be a constant in the range 0 to 3.
imm0_7: the parameter must be a constant in the range 0 to 7.
imm0_15: the parameter must be a constant in the range 0 to 15.
imm0_31: the parameter must be a constant in the range 0 to 31.
imm0_63: the parameter must be a constant in the range 0 to 63.
imm0_255: the parameter must be a constant in the range 0 to 255.
imm_n512_511: the parameter must be a constant in the range -512 to 511.
imm_n32_31: the parameter must be a constant in the range -32 to 31.
```

7.1 Intrinsics for DSP Control Register

```
void __builtin_mips_wrdsp (i32, imm0_63);
i32 __builtin_mips_rddsp (imm0_63);
i32 __builtin_mips_bposge32 ();
```

7.2 Intrinsics for Signed and Unsigned 8-bit Integers

```
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
i32 __builtin_mips_raddu_w_qb (v4i8);
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
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);
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (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);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
v4i8 __builtin_mips_absq_s_qb (v4i8); // DSPR2
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_shra_qb (v4i8, imm0_7); // DSPR2
v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7); // DSPR2
v4i8 __builtin_mips_shra_qb (v4i8, i32); // DSPR2
```

```
v4i8 __builtin_mips_shra_r_qb (v4i8, i32); // DSPR2
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8); // DSPR2
```

7.3 Intrinsics for Q15

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_absq_s_ph (v2q15);
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);
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);
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
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);
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
```

7.4 Intrinsics for Q31

```
q31 __builtin_mips_addq_s_w (q31, q31);
q31 __builtin_mips_subq_s_w (q31, q31);
q31 __builtin_mips_absq_s_w (q31);
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
```

```
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
q31 __builtin_mips_mulq_rs_w (q31, q31); // DSPR2
q31 __builtin_mips_mulq_s_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_r_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_r_w (q31, q31); // DSPR2
```

7.5 Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers

```
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16); // DSPR2
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);
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
```

7.6 Intrinsics for Mixed Data Types: Q15 and Q31

```
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
```

7.7 Intrinsics for 64-bit Accumulators

```
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_r_w (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_s_h (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);
```

7.8 Intrinsics for 32-bit Integers

```
i32 __builtin_mips_addsc (i32, i32);
```

```
i32 __builtin_mips_addwc (i32, i32);
i32 __builtin_mips_modsub (i32, i32);
i32 __builtin_mips_bitrev (i32);
i32 __builtin_mips_insv (i32, i32);
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
i32 __builtin_mips_append (i32, i32, imm0_31); // DSPR2
i32 __builtin_mips_balign (i32, i32, imm0_3); // DSPR2
i32 __builtin_mips_prepend (i32, i32, imm0_31); // DSPR2
a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);
```

7.9 Intrinsics for 16-bit Integers

```
v2i16 __builtin_mips_addu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16); // DSPR2
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16); // DSPR2
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, i32); // DSPR2
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16); // DSPR2
```

7.10 Intrinsics for Mixed DataTypes: 16-bit and 32-bit Integers

```
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31); // DSPR2
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31); // DSPR2
```

8 Experimental Results and Summary

The experimental results of the comparison in performance of four DSP kernels written in assembly and in C is shown in Table 1. As shown in the table, the C version of the IIR filter achieves the same performance as the hand-tuned assembly code. FIR, Complex FIR, and LMS kernels are slightly worse in the C versions than in the hand-tuned assembly code, but the differences are small, ranging from 6% to 12.98%.

Table 1 Results in Cycles of DSP Kernels

DSP Kernels	Hand-Tuned Assembly	Efficient C Code	Difference (%)
FIR	39	42	3 (7.69%)
IIR	32	32	0 (0.00%)
Complex FIR	2149	2278	129 (6.00%)
LMS	77	87	10 (12.98%)

This paper enables programmers to write efficient C code for the MIPS32 DSP ASE. Several tips and advanced topics are presented with examples and compiler-generated assembly code. Programmers are encouraged to migrate from assembly to C programming to enjoy faster time-to-market and still achieve high performance by using the MIPS32 DSP ASE support in compilers such as the the GCC compiler.

9 References

MIPS32® Architecture for Programmers Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture (MD00374)

For CodeSourcery users:

MIPS® SDE Library (MD00623)

MIPS® Toolchain Specifics (MD00624)

Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Tech, LLC, a Wave Computing company (“MIPS”) and MIPS’ affiliates as applicable. Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS or MIPS’ affiliates as applicable or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines. Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS (AND MIPS’ AFFILIATES AS APPLICABLE) reserve the right to change the information contained in this document to improve function, design or otherwise.

MIPS and MIPS’ affiliates do not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPSr3, MIPS32, MIPS64, microMIPS32, microMIPS64, MIPS-3D, MIPS16, MIPS16e, MIPS-Based, MIPSsim, MIPSpro, MIPS-VERIFIED, Aptiv logo, microAptiv logo, interAptiv logo, microMIPS logo, MIPS Technologies logo, MIPS-VERIFIED logo, proAptiv logo, 4K, 4Kc, 4Km, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KS, 4KSc, 4KSd, M4K, M14K, 5K, 5Kc, 5Kf, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 34K, 34Kc, 34Kf, 74K, 74Kc, 74Kf, 1004K, 1004Kc, 1004Kf, 1074K, 1074Kc, 1074Kf, R3000, R4000, R5000, Aptiv, ASMACRO, Atlas, "At the core of the user experience.", BusBridge, Bus Navigator, CLAM, CorExtend, CoreFPGA, CoreLV, EC, FPGA View, FS2, FS2 FIRST SILICON SOLUTIONS logo, FS2 NAVIGATOR, HyperDebug, HyperJTAG, IASim, iFlowtrace, interAptiv, JALGO, Logic Navigator, Malta, MDMX, MED, MGB, microAptiv, microMIPS, Navigator, OCI, PDtrace, the Pipeline, proAptiv, Pro Series, SEAD-3, SmartMIPS, SOC-it, and YAMON are trademarks or registered trademarks of MIPS and MIPS’ affiliates as applicable in the United States and other countries.

All other trademarks referred to herein are the property of their respective owners.

Copyright © Wave Computing, Inc. All rights reserved.
www.wavecomp.ai