Effective DSP Programming using MIPS® DSP Application Specific Extensions

This paper presents the MIPS32(R) DSP ASE (applications-specific extension) and provides performance optimization hints and example code. First, basic digital signal processing (DSP) terms, data types, and operations are defined. Next, the DSP ASE instruction set is introduced by classifying the new instructions into several classes and giving specific examples. Instruction timing information—important for achieving optimal performance—is summarized in concise table format. Several example DSP functions illustrate the transition from C code to MIPS32 assembly-language implementation and ultimately to DSP ASE implementation. The paper concludes by outlining useful code optimization guidelines.

Document Number: MD00475
Revision 01.22
June 30, 2008

Contents

1 Overview

The MIPS32® 24KE™ and 34K™ core families, collectively called 24KE throughout this white paper, implements the MIPS® DSP ASE (Digital Signal Processing Application Specific Extension). The DSP ASE enhances the MIPS32 and MIPS64 architectures with a set of new instructions and state that optimizes the performance of signal processing and multimedia applications. The performance of tasks such as audio encoding and decoding, image and video processing, and speech communication is improved by the use of the DSP ASE. The 24KE and the 34K core families implement the MIPS32 DSP ASE and therefore benefits the many signal processing application listed above.

This paper presents an overview of the DSP ASE and demonstrates the code optimizations possible with the new instruction set. The white paper is organized as follows:

- Section 2 “Introduction to Digital Signal Processing” provides a brief and basic introduction to the field of digital signal processing and its specific operations and requirements.
- Section 3 “The MIPS32® DSP ASE” outlines the capabilities of the DSP ASE as well as the supported data types and operations.
- Section 4 “Instruction Timing in the Pipeline of the 24KE™ and 34K™ Cores” provides instruction timing, result latencies, and other performance considerations, with most of the timing information summarized in tables.
- Section 5 “Programming Examples” provides several examples of DSP code optimized using the MIPS32 DSP ASE. It includes both C and regular MIPS32 assembly versions of the kernels in addition to the optimized DSP ASE version. Performance and memory use information is summarized for each algorithm. Some familiarity with C and assembly language programming for MIPS32 is required to understand the examples presented here.
- Section 6 “MIPS32® DSP ASE Optimization Guidelines” contains a collection of code optimization suggestions. These include performance enhancement methods applicable to MIPS32 and DSP code in general, as well as recommendations specific to the MIPS DSP ASE.
- Section 7 “Summary” provides a final summarization.

2 Introduction to Digital Signal Processing

2.1 What is DSP? What is a Signal?

In the digital signal processing area, the term signal is used to represent a sequence of data samples. These samples may have been obtained by either digitizing an analog waveform (e.g., a microphone input) or computed using an algorithm executed on a microprocessor or specialized hardware (e.g, video output from a DVD player). Signals can be one-dimensional, like the microphone input, or multi-dimensional--for example, the information contained in a digital picture can be considered a two-dimensional signal. Note that in the microphone input example the signal changes in time, while in the digital image case it changes with space (coordinates of each pixel). Thus we can classify the signals as being temporal (in the time-domain) or spatial.

Signals are usually uniformly sampled, meaning that the distance between every two consecutive samples in time or space is the same throughout the whole signal. The rate at which samples of the signal are obtained or produced is

called the sampling rate and is measured in samples/second or Hertz for time-domain signals. For example, common sampling rates used for audio processing are 44,100 samples/s and 48,000 samples/s.

Another important characteristic of a signal is the resolution, or number of bits, of each sample. If the number of bits per sample is **k**, the number of unique numbers representable in a sample is $\mathbf{2^k}$. This determines the range of values each sample can take, that is, the sample (or signal) resolution. The choice of the number of bits per sample is application-specific--for example, good-quality digital audio uses 16-bit samples while professional-grade audio uses 24 or more bits per sample. Image and video processing is usually performed on 8-bit or 10-bit samples.

Having defined the fundamental properties of signals, let us examine digital signal processing. Digital signal processing (DSP) is a rapidly expanding area of data processing methods and algorithms. These algorithms--often quite computationally intensive--allow analysis, synthesis, and transformation of digital signals of various origin and type. DSP allows signal transformations equivalent to the ones used in conventional analog systems--for example, standard-definition television--to be performed with better precision, lower noise, and higher reliability. Moreover, new applications such as HDTV, MP3, DVD, etc., have only become possible because of digital signal processing. Let us examine the basic goals pursued by DSP systems.

2.2 Typical Goals of a DSP Processing System

DSP systems accomplish one or more of the following signal processing goals:

1. **Extract or enhance desired features from the given signals.**
   Many applications ranging from medical image processing to sophisticated radar systems are in this category. The intent is to detect and measure specific signal components that are often partially masked by other interfering signal components.

2. **Suppress unwanted signal components.**
   This is a common goal that is often combined with the first one. Typical application examples include audio restoration and equalization, video noise suppression, etc.

3. **Convert signals into a different representation more suitable for further processing.**
   It is often valuable to convert the given signals into a different representation that makes certain operations easier to perform. Many such representations approximate the signals as sums of appropriately-chosen frequencies. The frequently used algorithm for accomplishing this conversion is the Fourier Transform. This algorithm is computationally intensive, but has efficient implementations--collectively know as Fast Fourier Transform (FFT) algorithms--that significantly reduce the number of operations needed to compute the result. Other algorithms, such as the Discrete Cosine Transform (DCT), are used in more specialized cases.

2.3 DSP Operations

An important aspect of many DSP algorithms is the use of the multiply-and-accumulate (MAC) operation. The simplest example is a digital filter that multiplies signal samples by filter coefficients and sums up the products to obtain a single output sample. This output sample is essentially a weighted average of the input signal samples; digital signal processors call this a convolution operation, while mathematicians call it a dot product. This process is repeated, with one new sample value used each time in the group of input signal samples. Alternatively, you can imagine the filter coefficients sliding over the signal samples with a new sample computed for each position of the filter coefficients. The following fragment of C code illustrates the algorithm:

```
for (k = 0; k < N; k++) {
    s = 0;
    for (i = 0; i < T; i++)
        s += input[k+i] * filter[i];
```

```
        output[k] = s;
    }
```

In this code example, N is the total number of samples that are processed and T is the number of filter coefficients (taps). The coefficients are stored in the array called filter in reverse order to simplify the array indexing. The innermost loop computes the dot product of the filter coefficients and the group of T input samples starting from sample k. Note the MAC operation at the heart of the code - one input sample is multiplied by one coefficient and the product is accumulated. Since MAC operations are very common in DSP algorithms, processors targeting DSP applications often provide specialized MAC instructions. The MIPS32 instruction set provides MAC instructions, while its capabilities are enhanced by the DSP ASE.

2.4 DSP Data Types

The example in the previous section does not make the data types explicit. It is easiest to perform all calculations using floating-point numbers, but hardware floating-point support is not always available. Even if it is, the performance of floating-point operations might be lower than that of integer operations and processors supporting floating-point units are typically bigger and have a higher cost. To avoid magnitude growth of the multiplication product, DSP algorithms often represent the data samples and the coefficients used in the computation as fractional numbers between -1 and +1. This is accomplished by placing the binary point separating the integer and fractional parts of the number immediately after the sign bit. Some of the commonly used fixed-point (or fractional) formats include Q15 (sign bit, zero integer bits, 15 fractional bits) and Q31 (sign bit, zero integer bits, 31 fractional bits). Sometimes data formats with one or more integer bits are used, trading off increased dynamic range for lower precision.

Note that the same hardware that performs integer add, subtract, and multiply operations can also be used for fixed-point operations. However, a fractional multiplication operation needs to be shifted left by one bit to re-align the data to a correct fractional double-width product. This is because multiplying two numbers in fixed-point format produces one redundant sign bit. The fractional multiply instructions in the MIPS32 DSP ASE automatically perform this correction.

2.5 Saturation And Rounding

Saturation and rounding are rare operations in non-DSP code, but used often in DSP algorithms. Saturation limits the result of a computation to the range implied by the input data type, thus avoiding overflow and underflow conditions. For example, an image processing application will typically use 8-bit unsigned integers for pixel representation, with the value of zero corresponding to black and the value of 255 corresponding to white.

Now imagine that we have to make an image 25% brighter. If we have a somewhat white pixel with a value of 220 and we multiply it by 1.25 to make it 25% brighter, we would get:

```
220 x 1.25 = 275
```

But 275 is not representable as an 8-bit number! Instead, the multiplication result will wrap-around to

```
275 - 256 = 19,
```

and suddenly we have a somewhat dark pixel! Saturation prevents cases like this by clipping the result to the maximum allowed value-- in this example, 255:

```
saturate(275) = 255
```

The rounding operation is used when the precision of some calculated result needs to be reduced. For example, multiplying Q15 numbers produces a Q31 result, but if we want to store it to memory as a 16-bit value, then we simply

throw away the low-order 16 bits. Simply discarding the lower bits is called truncation and is an easy way to reduce precision. A better way in many situations is to adjust the 16 high-order bits depending on the contents of the low-order bits. This operation, called rounding, increases the precision of the calculations and helps avoid problems like oscillation in some DSP algorithms. The MIPS32 DSP ASE includes both saturation and rounding options for the appropriate instructions.

Now that we have outlined the basic operations and data types used in digital signal processing applications, it is time to introduce the MIPS32 DSP application-specific extension.

3 The MIPS32® DSP ASE

3.1 What is the MIPS32® DSP ASE?

The MIPS32 DSP ASE is an optional architecture extension to the MIPS32 Release2 Architecture. This includes instructions and state tailored specifically for DSP applications. These instructions support data types and operations typically used in multimedia-oriented and general-purpose DSP algorithms. The DSP ASE uses the Single Instruction Multiple Data (SIMD) approach to process data packed into 32-bit registers as vectors of smaller-sized data. For example, four 8-bit or two 16-bit values can be packed into a single 32-bit register. Three additional 64-bit accumulators (for a total of four) have been added to facilitate the implementation of MAC-intensive algorithms. A new DSP Control Register has also been added; it has several status flags as well as few data fields used by some of the instructions.

The DSP ASE improves the performance of DSP algorithms. The speedup obtained over a MIPS32 implementation depends on the data types used, the type and frequency of DSP instructions in the code, and the algorithm. A speedup factor of 2x is typical as illustrated in Section 5 “Programming Examples”.

3.1.1 What is it Not?

The DSP extension to the MIPS32 architecture does not turn the MIPS-based™ RISC processor into a high-performance digital signal processor. The top-of-the-line devices manufactured by companies specializing in such high-end digital signal processors are very specialized and cost significantly more than a typical MIPS32 core with the DSP ASE.

The main goal of the DSP ASE is to give a boost to the DSP performance of the MIPS32 core without any significant clock speed degradation or area increase. It achieves that goal by utilizing clever design techniques that reuse the hardware that is already part of the core. Thus, the DSP ASE is a cost-efficient way to enhance performance, reduce power consumption, and integrate more signal-processing functionality into the application.

3.2 Instruction Classes And Supported Data Types

Each DSP ASE instruction has several variants that work with different data types or include rounding and saturation of the result. The desired data types and operation modes are encoded in the instruction mnemonic. Here is a brief list of the various instruction options:

- “Q” suffix in the root mnemonic
  Specifies that the instruction works on fractional (fixed-point) data. For example, MULQ performs fractional multiplication.

- “U” suffix in the root mnemonic
  Specifies unsigned data type. For example, ADDU performs unsigned addition.

- “E” suffix in the root mnemonic
  Used to specify that the instruction produces expanded-precision results, i.e., the number of bits in the result is larger than the bit width of the input data.

- “R” suffix in the root mnemonic
  Used to specify that the instruction produces reduced-precision results, i.e., the number of bits in the result is smaller than the bit width of the input data

- “_R” suffix - selects rounding mode

- “_S” suffix - selects saturation mode

- “_RS” suffix - specifies rounding and saturation

- “_SA” suffix - requests accumulator saturation

- “.W” type selector - single 32-bit word

- “.PH” type selector - two packed 16-bit halfwords

- “.QB” type selector - four packed 8-bit bytes

For example, the MULQ_RS.PH rd,st,rt instruction multiplies two 16-bit fixed-point numbers stored in the two halves of the first input register by the two 16-bit fixed-point numbers stored in the second input register. It rounds and saturates the two results before writing them to the destination register.

Note that the instruction options can not be used in any combination. Please consult the “DSP ASE Architecture for Programmers” Reference Manual (MD00374) for information about the instruction variants supported by the MIPS32 DSP ASE. The DSP ASE instructions are also described in Chapter 7 of the “Programming the MIPS32® 24KE Core Family” Manual (MD00458).

3.3 The Instruction Set

The MIPS32 DSP ASE instructions can be classified depending on the implemented function. These classes are listed including some important instruction examples.

3.3.1 Arithmetic Class

The instructions in this class perform basic arithmetic operations such as addition and subtraction, as well as some more specialized operations like precision reduction (data packing) and precision expansion (data unpacking). For example:

- The ADDQ.PH and ADDQ_S.PH instructions compute the two 16-bit sums of the corresponding 16-bit fixed-point data values and perform optional saturation. Similar instructions are provided for 32-bit fixed-point words and packed unsigned 8-bit bytes. There is also an add with carry instruction (ADDWC); the carry flag resides in the DSPControl Register and is set by another special instruction (ADDSC).

- The ABSQ instruction computes the absolute value of the elements of the input register.

- The RADDU instruction performs horizontal (reduction) sum of the source register data elements.
- The MODSUB instruction can be used to implement circular buffers, i.e., arrays with indexes that wrap-around after each pass through the array.

3.3.2 Shift Class

The instructions in the shift class implement logical and arithmetic shift of the individual SIMD data elements that are packed into 32-bit registers. The shift amount can be variable (specified by a register) or fixed. The saturation option in the left shift instructions, e.g., SHLL_S.PH, ensures that the result is clipped to the appropriate range in the case of signed overflow.

3.3.3 Multiply Class

Instructions in this category include:

- Integer and fractional element-wise multiplication (e.g., MULEU_S.PH and MULQ_RS.PH)
- Integer and fractional dot product accumulate and subtract (DPAU, DPSU, DPAQ_S, DPSQ_S), and other operations including instructions producing 64-bit results.
- The MULSAQ_S.W.PH and DPAQ_S.W.PH instructions compute the real and imaginary parts of a complex multiply, accumulating the results to a specified accumulator. This is useful for implementing complex-coefficient FIR filters.

Some of the instructions, like DPAQ_SA.L.W, use the "_SA" suffix to indicate that the saturation is performed in two steps. In the first step the input operands are checked for the -1 x -1 special case and the result is clamped if necessary. In the second step the accumulator is saturated after accumulation of the multiplication product.

3.3.4 Bit Manipulation Class

- The bit-reversal instruction (BITREV) reverses the order of the 16 right-most bits in the source register. Bit-reversed addressing modes are often used with Fast Fourier Transform (FFT) algorithms.
- The INSV instruction extends the MIPS32 INS instruction to support variable position and size values, which are obtained from the DSPControl Register.
- The data replication instructions REPL copy a given scalar value to all the SIMD elements of the destination register.

3.3.5 Compare and Pick Class

- The first instruction in this class performs SIMD (element-wise) comparison of the data in the source registers. The result of the comparison can be written either to a special field of the DSPControl Register (e.g., CMP.LT.PH) or to one of the general- purpose registers (e.g., CMPGU.LE.QB). \
- After the CMP instruction variants, the PICK.QB and PICK.PH instructions can be used to select elements from two registers based on the result of the comparison.
- The last instruction in this class, PACKRL.PH, takes the right halfword from the first source register and the left halfword from the second register, exchanges their places, and packs them into one register.

3.3.6 Accumulator and DSPControl Register Access Class

- This class contains the EXTR instruction that extract values from the accumulator after shifting it to the right. The EXTR instruction extracts the 32 least significant bits after the shift to the right. The EXTR instruction has rounding and saturation variants and supports a variable shift amount (specified in a register).

- The EXTP instruction extracts a number of bits from a specified position. The size of the field to extract is encoded in the instruction or set by a register. The position comes from a field in the DSPControl Register. A variant of the instruction, EXTPDP, decrements the value of the position field. When used in conjunction with the MTHLIP and BPOSGE32, this instruction can be used to implement efficient bitstream decoding.

- The MTHLIP instruction shifts the 64-bit accumulator left by 32 bits, loads the data value from the source register into the 32 least significant bits of the accumulator and increments the position field in the DSPControl Register by 32. This operation is typically used to load a new word from the input bit stream into accumulator. This is doe when the current extract position is less than 32, that is, in the LO part of the HI/LO pair. This state is checked by the BPOSGE32 instruction, which branches to the target as long as the value of the position field in DSPControl is still greater than or equal to 32.

- The SHILO instruction does a logical shift of the whole 64-bit accumulator left or right by the specified shift amount.

- The WRDSP and RDDSP instructions transfer data between the general-purpose registers and the DSPControl Register.

3.3.7 Indexed Load Class

The instructions in this category add a new addressing mode for loading integer and fixed-point data. The address is specified in the form of base+index, where both the base address and the index (offset) come from registers. The three variants, LBUX, LHX, and LWX load unsigned bytes, half-words, and 32-bit words respectively.

4 Instruction Timing in the Pipeline of the 24KE™ and 34K™ Cores

This chapter presents the instruction timing information for the MIPS 24KE and 34K cores. With respect to instruction timing the two cores are identical and are referred to as 24KE in the following sections. The timing information, which includes result latencies, instruction issue rates, interlocks, etc., is very important for optimizing the performance of critical DSP code. It should be underlined that mapping a DSP algorithm to DSP ASE instructions is not the final step of the optimization process; the code still needs to be scheduled, meaning that the instructions have to be rearranged in a way that maximizes the performance by reducing the pipeline stalls. This is where the instruction timing information becomes important.

It should also be noted that optimizing the code by selecting an instruction sequence and then scheduling it according to the instruction timing is an iterative process. For example, attempting to schedule a code fragment may reveal that additional loop unrolling and software pipelining is necessary in order to mask all stalls in a loop body. Both the loop unrolling and the software pipelining methods for improving the performance are discussed in the “Optimization Guidelines” chapter.

4.1 Pipelining

Before presenting the detailed instruction timing information, let’s review some pipeline terms and concepts. All modern high-performance processors use pipelined execution control in order to improve their performance. The pipeline consists of several stages and each instruction proceeds through all stages during its execution. All pipeline stages are controlled by a common clock signal, which defines the clock cycle time of the processor. The benefit of using a pipeline is that several instructions (as many as there are pipeline stages) are overlapped inside the pipeline. For example, if the first instruction from a given code sequence has reached stage 5, the next instruction is in stage 4, the 3rd instruction is stage 3, and so on. Thus, even though it takes several cycles to execute a single instruction, the effective throughput is one instruction per cycle.

The previous discussion outlined the ideal case. In practice, things do not run so smoothly. Since the instructions flowing through the pipeline are overlapped, one instruction may need data that the previous instruction has not produced yet. For example, the current instruction may be trying to use the value of register $v0, which the previous instruction computes--a case known as register dependency. If the current instruction is allowed to proceed, it will use some old data and produce incorrect (compared to the intuitive one-at-a-time instruction execution) result. Hence the processor needs to detect such hazards and prevent them from affecting the result of the operation.

There are two ways to handle the data hazards. The first one is to forward the result from the previous instruction directly to the current one, bypassing the register file update. This is possible for most arithmetic and logical instructions. The update of the register file happens in parallel, so the next instruction can use the already updated register value. Unfortunately, it is not possible to perform forwarding in all cases. The second method to ensure correct operation is to stall the pipeline by stopping the execution of the current instructions and all instructions that follow it, but allowing the previous instruction to complete. After it completes the pipeline resumes normal operation. Stalling the pipeline for one or more cycles creates “bubbles” in the pipeline: no instruction completes during these stall cycles. Hence stall cycles reduce the performance and it is important to arrange the instructions in a way that avoids them.

The MIPS 24KE family of cores has an 8-stage integer (ALU) pipeline and a separate 5-stage pipeline for the multiply/divide unit (MDU). The MDU pipeline works in parallel to the integer pipeline and does not stall when the integer pipeline does. Appropriate forwarding is provided inside the ALU pipeline and between the ALU and MDU pipelines.

4.2 Timing Information

We are not going to discuss the operation of the 24KE pipelines in detail. Instead, the information presented in the following tables and the accompanying notes summarizes the instruction timing information needed by an assembly language programmer trying to extract the best possible performance. Table 1 and Table 2 define several instruction categories for the MIPS32 and the DSP ASE instruction sets, respectively. A short description as well as a list of instructions are provided for each category.

Table 1 MIPS32 Instruction Categories

Category	Description	Instructions[1]
LD	Load instructions	LB, LBU, LH, LHU, LL, LW, LWPC, LWL, LWR
ST	Store instructions	SB, SC, SH, SW, SWL, SWR
MUL-ACC	Multiply instructions updating the accumulator	MADD, MADDU, MSUB, MSUBU, MULT, MULTU
MUL-GPR	Multiply instructions updating a GPR register	MUL
DIV	Divide instructions updating the accumulator	DIV, DIVU
ACC-READ	Accumulator reading instructions	MFHI, MFLO
ACC-WRITE	Accumulator writing instructions	MTHI, MTLO
ALU	MIPS32 ALU instructions	ADD, ADDI, ADDIU, ADDU, AND, ANDI, CLO, CLZ, EXT, INS, LUI, MOVN, MOVZ, NOR, OR, ORI, ROTR, ROTRV, SEB, SEH, SLL, SLLV, SLT, SLTI, SLTIU, SLTU, SRA, SRAV, SRL, SRLV, SUB, SUBU, WSBH, XOR, XORI

1. This table lists only integer data-processing instructions. Excluded from the list are floating-point, MIPS16™, coprocessor, cache control, prefetch, trap, and other instructions typically used by system software.

Table 2 DSP ASE Instruction Categories

Category	Description	Instructions[1]
LD	Additional load instructions	LBUX, LHX, LWX
DSP-MAC	MAC instructions not saturating the accumulator	DPAQ_S, DPAU, DPSQ_S, DPSU, MAQ_S, MULSAQ
DSP-MAC-SAT	MAC instructions that saturate the accumulator	DPAQ_SA, DPSQ_SA, MAQ_SA
DSP-MUL-GPR	DSP multiply instructions that update a GPR register	MULQ, MULEQ, MULEU
DSP-ACC-EXT	Accumulator extract instructions	EXTP, EXTPDP, EXTPDPV, EXTPV, EXTR, EXTRV
DSP-ACC-MOD	Accumulator modify instructions	MTHLIP, SHILO, SHILOV
DSP-ALU	DSP ALU instructions	ABSQ, ADDQ, ADDSC, ADDU, ADDWC, BITREV, CMP, CMPGU, CMPU, INSV, MODSUB, PACKRL, PICK, PRECEQ, PRECEQU, PRECEU, PRECRQ, PRECRQU, RADDU, RDDSP, REPL, REPLV, SHLL, SHLLV, SHRA, SHRAV, SHRL, SHRLV, SUBQ, SUBU, WRDSP

1. Most of the DSP ASE instructions have variants operating on different data types, applying rounding and saturation, etc. Only the root stem of the mnemonic is listed in the table except for instructions classified in different categories depending on their variants.

Table 3 shows the result delay (possible stall cycles) of executing a pair of instructions from two categories. The first

instruction is used to select the row, and the second instruction is used to select the column. The information in the table assumes data dependency between the two instructions--either through a GPR register or through a HI/LO accumulator pair. Instruction sequences without data dependency do not stall unless specifically noted. The dash in some of the cells in the table specifies that no data dependency is possible; hence the two instructions will always run without stall cycles.

Note that the repeat rate of most instructions without data dependency through a GPR register is one per cycle. The accumulator-saturating MAC instructions have a repeat rate of one instruction every two cycles if the second instruction updates the same accumulator.

Table 3 MIPS 24KE Pipeline Delays

		Second Instruction												
		LD/ST	MUL-ACC	MUL-GPR	DIV	ACC-READ	ACC-WRITE	ALU	DSP-MAC	DSP-MAC-SAT	DSP-MUL-GPR	DSP-ACC-EXT	DSP-ACC-MOD	DSP-ALU
First Instruction	LD	0/2[1]	1	1	1	-	1	1	1	1	1	1	1	1
	ST	-	-	-	-	-	-	-	-	-	-	-	-	-
	MUL-ACC	-	0	-	-	0	-	-	0	0	-	3	3	-
	MUL-GPR[2]	4/5[3]	4	4	4	-	4	4	4	4	4	4	4	4
	DIV[4]	-	9-35	-	12-38	7-33	7-33	-	9-35	9-35	-	11-37	11-37	-
	ACC-READ[2]	4/5[3]	4	4	4	-	4	4	4	4	4	4	4	4
	ACC-WRITE	-	1	-	-	0	-	-	1	1	-	3	3	-
	ALU	0/1[3]	0	0	0	-	0	0	0	0	0	0	0	0
	DSP-MAC	-	0	-	-	0	-	-	0	0	-	3	3	-
	DSP-MAC-SAT	-	1	-	-	0	-	-	1	1	-	3	3	-
	DSP-MUL-GPR[2]	4/5[3]	4	4	4	-	4	4	4	4	4	4	4	4
	DSP-ACC-EXT[2]	4/5[3]	4	4	4	-	4	4	4	4	4	4	4	4
	DSP-ACC-MOD	-	1	-	-	0	-	-	1	1	-	3	3	-
	DSP-ALU	1/2[3]	1	1	1	-	1	1	1	1	1	1	1	1[5]

1. There will be two stall cycles if the result of a load is used for address calculation of a second load. Load timing assumes cache hits.
2. MDU instructions that update GPR registers compete for access to the register file with ALU instructions. Hence it may take more cycles than indicated here to write the MDU result to the destination GPR register. See 4.4 “MDU/ALU GPR Write Port Contention” on page 14.
3. The larger number of stall cycles applies when the dependency is on register used for address calculation in the load/store instruction.
4. The exact number of cycles depends on the type of instruction (DIV or DIVU) and the size of the first operand. Note that some of the divide instruction sequences are rarely used in practice and their timing is not guaranteed.
5. Several special instruction pairs execute with zero delay cycles. These include ADDSC/ADDWC, CMP/PICK, and WRDSP/INSV.

4.3 Example Usage

Let's try using the timing information tables with a simple example. The code below loads four 16-bit data values packed into two 32-bit words, performs SIMD addition and SIMD multiplication, and stores the result. No scheduling of the code has been performed yet.

```
addiu       $a0, $a0, 8          # ALU
lw          $t0, 0($a0)          # LD/ST
lw          $t1, 4($a0)          # LD/ST
addq_ph     $t2, $t0, $t1        # DSP-ALU
mulq_rs.ph  $v0, $t2, $t3        # DSP-MUL-GPR
sw          $v0, 0($a1)          # LD/ST
```

Note that we have already identified the category of each of the above instructions using tables 1 and 2. Next, using Table 3 we find out that there will be one stall cycle between the addiu and the first lw instruction since register $a0, which is updated by the addiu instruction, is used as an address in the lw instruction. The two load instructions execute back to back (no stall cycles), but the addq instruction will stall for one cycle waiting for the loaded data. Similarly, the mulq instruction will stall for one cycle waiting for the addq result to become available. And finally, the store instruction will stall for four cycles until the multiplication result is ready. In other words, the example above will execute like this:

```
addiu       $a0, $a0, 8          # ALU
  # 1 stall cycle - address modified
lw          $t0, 0($a0)          # LD/ST
lw          $t1, 4($a0)          # LD/ST
  # 1 stall cycle - load delay
addq_ph     $t2, $t0, $t1        # DSP-ALU
  # 1 stall cycle - result dependency on DSP ALU instruction
mulq_rs.ph  $v0, $t2, $t3        # DSP-MUL-GPR
  # 4 stall cycles - result dependency on DSP MDU instruction
sw          $v0, 0($a1)          # LD/ST
```

Clearly, this snippet of code will not deliver optimal performance.

4.4 MDU/ALU GPR Write Port Contention

Note that, as stated in the footnotes of Table 3, MDU instructions producing result in a GPR register compete for access to the register file with ALU instructions. This interaction is rarely visible, but can have negative performance impact if not taken into account. Consider the following instruction sequence:

```
lw          $t2, 0($a0)
lw          $t3, 4($a0)
mulq_rs.ph  $s0, $t0, $t2        # (1)
mulq_rs.ph  $s1, $t1, $t3        # (2)
rotr        $t0, $t0, 16
rotr        $t1, $t1, 16
addiu       $a0, $a0, 8
addiu       $a1, $a1, 8
addq.ph     $s4, $s0, $s1        # (3)
subq.ph     $s5, $s0, $s1        # (4)
sw          $s4, 0($a1)
sw          $s5, 4($a1)
```

The two multiplication results from the instructions marked (1) and (2) are needed by the addq (3) and subq (4) instructions. According to the timing information presented in the tables above, the delay from a multiply instruction updating a GPR register to an ALU instruction using that register is four cycles. This means that the first and second mulq results should be ready for the addq instruction (3) and the following subq instruction (4). However, note that every instruction between the mulq instructions and the addq is an ALU instruction that writes to a GPR register. Hence the multiplication results are queued and wait for an opportunity to get written to the corresponding GPR registers--$s0 and $s1. When the addq instruction is executed, the pipeline is stalled to allow $s0 and $s1 to be updated, and then the addq proceeds with the updated values. This is illustrated here:

```
lw          $t2, 0($a0)
lw          $t3, 4($a0)
mulq_rs.ph  $s0, $t0, $t2        # (1)
mulq_rs.ph  $s1, $t1, $t3        # (2)
rotr        $t0, $t0, 16
rotr        $t1, $t1, 16
addiu       $a0, $a0, 8
addiu       $a1, $a1, 8
  # 1 stall cycle - register $s0 updated with 1st mulq result
  # 1 stall cycle - register $s1 updated with 2nd mulq result
addq.ph     $s4, $s0, $s1        # (3)
subq.ph     $s5, $s0, $s1        # (4)
sw          $s4, 0($a1)          # (6)
sw          $s5, 4($a1)          # (7)
```

The way to properly schedule this code and avoid the stalls is to insert some instructions that to not immediately update GPRs between the mulq instructions and the addq. In this example the two sw instructions (6) and (7) can be moved up, before the addq, if the code fragment is software pipelined. This way while the sw instructions store the previous iteration results to memory, the mulq results will update the GPR registers. The modified code runs without stalls:

```
lw          $t2, 0($a0)
lw          $t3, 4($a0)
mulq_rs.ph  $s0, $t0, $t2        # (1)
mulq_rs.ph  $s1, $t1, $t3        # (2)
rotr        $t0, $t0, 16
rotr        $t1, $t1, 16
addiu       $a1, $a1, 8
addiu       $a0, $a0, 8
sw          $s4, -8($a1)         # (6) $s0 updated here
sw          $s5, -4($a1)         # (7) $s1 updated here
addq.ph     $s4, $s0, $s1        # (3)
subq.ph     $s5, $s0, $s1        # (4)
```

Note that the two addiu instructions updating $a0 and $a1 have been exchanged in order to avoid a stall due to address register update between the addiu instruction modifying $a1 and the sw instruction using it as a pointer.

Other mulq instructions can also be used as placeholders for GPR updates since the mulq instructions do not update the destination GPR immediately and the GPR write port is free for other use.

4.5 Branches

Finally, some timing information about branches. Pipelined microprocessors suffer from large branch penalties since several partially executed instructions that are already in the pipeline have to be nullified if the branch is taken. To reduce this penalty modern processors employ static and dynamic branch prediction and if the prediction is correct, the penalty can be reduced. Additionally, some processors, including the MIPS 24KE, always execute the instruction

in the branch delay slot (with the exception of the branch likely instructions, which are ignored here). A useful instruction should be placed in the branch delay slot since its execution is independent of whether the branch is actually taken or not.

The MIPS 24KE core implements static and dynamic branch prediction. If the prediction is correct, which happens most of the time especially for simple signal processing loops, there is no branch penalty. In case the branch is incorrectly predicted, irrespective of whether or not the branch was taken or not taken, the MIPS 24KE core inserts four stall cycles after the branch delay instruction (which is always executed). Similarly, the MIPS 34K core has a 5-cycle branch mispredict penalty. Note that it is not possible to hide the mispredicted branch penalty by placing independent instructions after the branch similarly to what is usually done to cover stalls due to data dependencies.

In rare cases, when several branches are taken in rapid succession, the instruction buffer of the 24KE core may run out of instructions. In this case even correctly predicted branches may incur up to two stall cycles.

5 Programming Examples

5.1 Example DSP Kernels

The code examples in this chapter illustrate the advantage of using the MIPS32 DSP ASE instructions. Three typical DSP algorithms, dot product, complex-valued finite impulse response (FIR) filter, and vector maximum are presented. The first two algorithms are implemented in four different ways.

1. First, a floating-point C version demonstrates the data flow of the algorithm.

2. Next, the algorithm is implemented again in C, but this time using fractional (fixed-point) Q15 numbers. Rounding, scaling, and normalization of the multiplication results have to be performed explicitly.

3. The MIPS32 assembly language version follows the fixed-point C version and is useful for performance estimation for cores that do not implement the DSP ASE.

4. And finally, the DSP ASE assembly language version uses the new instructions with SIMD capabilities and built-in rounding and saturation to significantly reduce the cycle count.

The third algorithm, the vector maximum operates on 8-bit unsigned numbers (bytes) and hence has only one C implementation. Both a MIPS32 and a DSP ASE assembly language versions are provided. The data type used by this algorithm makes it possible to use 4-way SIMD, which greatly improves the performance of the DSP ASE version.

The cycle counts for the assembly versions of the algorithms assume best execution environment conditions, i.e., warm caches and pre-loaded branch prediction tables. This is a common situation for DSP applications, where the same functions are repeatedly executed and the output data from one function becomes input data for another function.

Note that to keep the examples small and understandable, only the computational kernel code is shown. Function entry and exit, stack and memory management, argument passing, etc. is omitted for clarity. This extra code will be similar across the different versions of the same algorithm and would not significantly affect the performance comparison.

Note that the illustrated code was chosen as a compromise between clarity and performance. Using more aggressive optimization techniques the performance can be improved even further. Some of these techniques are described in Section 6 “MIPS32® DSP ASE Optimization Guidelines”.

5.2 Example 1: Dot Product

The dot product is a central operation in digital FIR filters. A set of samples from one signal are multiplied by the corresponding samples from another signal. The products are accumulated to produce a single output result. In digital filters the first set of samples is a subset of the input data while the second one is the filter coefficients. Each dot product calculation produces one output value of the filter. The range of samples from the input signal is offset (moved) by one sample before the calculation of the next output sample.

5.2.1 Floating-point C version

```
float a[N], b[N];
float s = 0.0;

for (i = 0; i < N; i++)
    s += a[i] * b[i];
```

5.2.2 C version using Q15 Data Type

```
#define SCALE 5

short a[N], b[N];               // data arrays in Q15 format
long long s = 0;                // 64-bit accumulator
short r;                        // 16-bit result

for (i = 0; i < N; i++)
    s += a[i] * b[i];           // accumulate Q1.30 products

s = s >> SCALE;                 // scaling

if (s > 0x7FFF)                 // positive saturation?
    r = 0x7FFF;                 // clamp the result to +1
else if (s < -0x8000)           // negative saturation?
    r = -0x8000;                // clamp the result to -1
else                            // no saturation
    r = s & 0xFFFF;             // discard redundant sign bits
```

5.2.3 MIPS32® Assembly Language Version using Q15 Data Type

Performance: 15*N/4 + 19 = 394 cycles for N = 100
Memory use: 29 instructions = 116 bytes

```
# $a0 - pointer to a[]
# $a1 - pointer to b[]
# $a2 - N (multiple of 4)
# $v0 - result

sll    $a2, $a2, 1              # multiply N by sizeof(short)
add    $a3, $a0, $a2            # compute final address in a[]

mult   $zero, $zero             # zero the accumulator
```

```
loop:                           # the loop is unrolled 4x

lh      $t0, 0($a0)             # load 1st element from a[]
lh      $t1, 0($a1)             # load 1st element from b[]
lh      $t2, 2($a0)             # load 2nd element from a[]
lh      $t3, 2($a1)             # load 2nd element from b[]
lh      $t4, 4($a0)             # load 3rd element from a[]
lh      $t5, 4($a1)             # load 3rd element from b[]
lh      $t6, 6($a0)             # load 4th element from a[]
lh      $t7, 6($a1)             # load 4th element from b[]

madd    $t0, $t1                # multiply and accumulate 1st elements
madd    $t2, $t3                # multiply and accumulate 2nd elements
madd    $t4, $t5                # multiply and accumulate 3rd elements
madd    $t6, $t7                # multiply and accumulate 4th elements

addiu   $a0, $a0, 8             # increment a[] pointer by 4*sizeof(short)

bne     $a0, $a3, loop          # loop until the end of the 1st input array
addiu   $a1, $a1, 8             # increment b[] pointer by 4*sizeof(short)

# 4 stall cycles on the 24K/24KE - mispredicted branch

mflo    $v0                     # get 32 accumulator LSBs
mfhi    $v1                     # get 32 accumulator MSBs

addiu   $t3, $zero, 0x7FFF      # load upper limit 0x00007FFF
addiu   $t4, $zero, 0x8000      # load lower limit 0xFFFF8000 (sign extended)

# 1 stall cycle - result latency of mflo

srl     $v0, $v0, SCALE         # scale the 32 LSBs of the result
sll     $t2, $v1, 32-SCALE      # isolate MSBs to be combined with the LSBs
or      $v0, $v0, $t2           # combined scaled 32-bit result

slt     $t2, $t3, $v0           # set $t2 if result larger than 0x7FFF
movn    $v0, $t3, $t2           # positive clipping to 0x7FFF if $t2 set
slt     $t2, $v0, $t4           # set $t2 if result smaller than 0xFFFF8000
movn    $v0, $t4, $t2           # negative clipping to 0xFFFF8000 if $t2 set
```

5.2.4 DSP ASE Assembly Language Version using Q15 Data Type

Performance: 2*N + 10 = 210 cycles for N = 100
Memory use: 14 instructions = 56 bytes

```
# $a0 - pointer to a[]
# $a1 - pointer to b[]
# $a2 - N (multiple of 4)
# $v0 - result

sll     $a2, $a2, 1             # multiply N by sizeof(short)
add     $a3, $a0, $a2           # compute final address in a[]

sub     $s0, $a1, $a0           # compute pointer differences: use LWX
addiu   $s1, $s0, 4             #   to avoid updating two pointers
```

```
mult   $zero, $zero             # zero the accumulator

loop:                           # the loop is unrolled 4x

lw     $t0, 0($a0)              # load 1st and 2nd elements from a[]
lwx    $t1, $s0($a0)            # load 1st and 2nd elements from b[]
lw     $t2, 4($a0)              # load 3rd and 4th elements from a[]
lwx    $t3, $s1($a0)            # load 3rd and 4th elements from b[]

addiu  $a0, $a0, 8              # increment a[] pointer by 4*sizeof(short)

dpaq_s.w.ph  $ac0, $t0, $t1     # MAC 1st and 2nd elements

bne    $a0, $a3, loop           # loop until the end of the 1st input array
dpaq_s.w.ph  $ac0, $t2, $t3     # MAC 3rd and 4th elements

# 4 stall cycles on the 24K/24KE - mispredicted branch

extr_s.h      $v0, $ac0, SCALE  # scale and saturate the result

# 4 stall cycles if the result is immediately used - extr_s result latency
```

5.3 Example 2: Complex-Value FIR Filter

The previous example illustrated a real-value dot product, which is the kernel of real-value FIR filters. This example shows a complex-value FIR filter; it operates on complex input data and has complex coefficients. Each complex number--an input sample or a coefficient--has real and imaginary parts. Four multiplication products are computed and added/subtracted to calculate the real and imaginary parts of the complex multiply operation.

5.3.1 Floating-point C version

```
typedef struct {
    float re;
    float im;
} complex;

complex x[N+K-1], h[K], r[N];

for (i = 0; i < N; i++) {
    complex s;
    s.re = s.im = 0.0;

    for (j = 0; j < K; j++) {
        complex a = x[i+j];
        complex b = h[j];
        s.re += (a.re * b.re) - (a.im * b.im);
        s.im += (a.re * b.im) + (a.im * b.re);
    }
    r[i] = s;
}
```

5.3.2 C version using Q15 Data Type

```
typedef struct {
    short re;
```

```
    short im;
} complex;

#define SCALE 5
complex x[N+K-1], h[K], r[N];

for (i = 0; i < N; i++) {
    complex s;
    long long sRe = 0;
    long long sIm = 0;

    for (j = 0; j < K; j++) {
        complex a = x[i+j];
        complex b = h[j];
        sRe += (a.re * b.re) - (a.im * b.im);
        sIm += (a.re * b.im) + (a.im * b.re);
    }
    sRe = sRe >> SCALE;          // scaling
    if (sRe > 0x7FFF)            // positive saturation?
        s.re = 0x7FFF;           // clamp the result to +1
    else if (sRe < -0x8000)      // negative saturation?
        s.re = -0x8000;          // clamp the result to -1
    else                         // no saturation
        s.re = sRe & 0xFFFF;     // discard redundant sign bits

    sIm = sIm >> SCALE;          // scaling
    if (sIm > 0x7FFF)            // positive saturation?
        s.im = 0x7FFF;           // clamp the result to +1
    else if (sIm < -0x8000)      // negative saturation?
        s.im = -0x8000;          // clamp the result to -1
    else                         // no saturation
        s.im = sIm & 0xFFFF;     // discard redundant sign bits

    r[i] = s;
}
```

5.3.3 MIPS32® Assembly Language Version using Q15 Data Type

Performance: 15*N*K + 36*N + 11 = 51,611 cycles for N = 100, K = 32
Memory use: 65 instructions = 260 bytes

```
# $a0 - pointer to x[]
# $a1 - pointer to h[]
# $a2 - pointer to r[]
# $s0 - N (number of samples)
# $s1 - K (number of filter taps, multiple of 2)

sll    $s0, $s0, 2              # multiply N by sizeof(complex)
sub    $s2, $s0, 4              # set $s2 to (N-1)*sizeof(complex)
add    $s0, $s0, $a2            # compute final address in r[]

sll    $s1, $s1, 2              # multiply K by sizeof(complex)
add    $a3, $a1, $s1            # compute final address in h[]

addiu  $t5, $zero, 0x7FFF       # load upper limit 0x00007FFF
addiu  $t6, $zero, 0x8000       # load lower limit 0xFFFF8000 (sign extended)
```

```
loop:

mult   $zero, $zero             # zero accumulator ac0

loopRe:                         # compute real part - loop unrolled 2x

lh     $t0, 0($a0)              # load a.re from x[]
lh     $t1, 2($a0)              # load a.im from x[]
lh     $t2, 0($a1)              # load b.re from h[]
lh     $t3, 2($a1)              # load b.im from h[]

madd   $t0, $t2                 # ac0 += a.re * b.re
msub   $t1, $t3                 # ac0 -= a.im * b.im

lh     $t0, 4($a0)              # load next a.re from x[]
lh     $t1, 6($a0)              # load next a.im from x[]
lh     $t2, 4($a1)              # load next b.re from h[]
lh     $t3, 6($a1)              # load next b.im from h[]

madd   $t0, $t2                 # ac0 += a.re * b.re
msub   $t1, $t3                 # ac0 -= a.im * b.im

addiu  $a1, $a1, 8              # increment h[] pointer by 2*sizeof(complex)

bne    $a1, $a3, loopRe
addiu  $a0, $a0, 8              # increment x[] pointer by 2*sizeof(complex)

# 4 stall cycles on the 24K/24KE - mispredicted branch

mflo   $v0                      # get 32 accumulator LSBs
mfhi   $v1                      # get 32 accumulator MSBs

sub    $a0, $a0, $s1            # point $a0 back to the start position in x[]
sub    $a1, $a1, $s1            # point $a1 back to the beginning of h[]

mult   $zero, $zero             # zero accumulator ac0

srl    $v0, $v0, SCALE          # scale the 32 LSBs of the result
sll    $t2, $v1, 32-SCALE       # isolate MSBs to be combined with the LSBs
or     $v0, $v0, $t2            # combined scaled 32-bit result

slt    $t2, $t5, $v0            # set $t2 if result larger than 0x7FFF
movn   $v0, $t5, $t2            # positive clipping to 0x7FFF if $t2 set
slt    $t2, $v0, $t6            # set $t2 if result smaller than 0xFFFF8000
movn   $v0, $t6, $t2            # negative clipping to 0xFFFF8000 if $t2 set

sh     $v0, 0($a2)              # store the real part

loopIm:                         # compute imaginary part - loop unrolled 2x

lh     $t0, 0($a0)              # load a.re from x[]
lh     $t1, 2($a0)              # load a.im from x[]
lh     $t2, 0($a1)              # load b.re from h[]
lh     $t3, 2($a1)              # load b.im from h[]

madd   $t1, $t2                 # ac0 += a.im * b.re
madd   $t0, $t3                 # ac0 += a.re * b.im
```

```
lh      $t0, 4($a0)             # load next a.re from x[]
lh      $t1, 6($a0)             # load next a.im from x[]
lh      $t2, 4($a1)             # load next b.re from h[]
lh      $t3, 6($a1)             # load next b.im from h[]

madd    $t1, $t2                # ac0 += a.im * b.re
madd    $t0, $t3                # ac0 += a.re * b.im

addiu   $a1, $a1, 8             # increment h[] pointer by 2*sizeof(complex)

bne     $a1, $a3, loopIm
addiu   $a0, $a0, 8             # increment x[] pointer by 2*sizeof(complex)

# 4 stall cycles on the 24K/24KE - mispredicted branch

mflo    $v0                     # get 32 accumulator LSBs
mfhi    $v1                     # get 32 accumulator MSBs

sub     $a0, $a0, $s2           # point $a0 to the new start position in x[]
sub     $a1, $a1, $s1           # point $a1 back to the beginning of h[]
addiu   $a2, $a2, 4             # increment r[] pointer by sizeof(complex)

srl     $v0, $v0, SCALE         # scale the 32 LSBs of the result
sll     $t2, $v1, 32-SCALE      # isolate MSBs to be combined with the LSBs
or      $v0, $v0, $t2           # combined scaled 32-bit result

slt     $t2, $t5, $v0           # set $t2 if result larger than 0x7FFF
movn    $v0, $t5, $t2           # positive clipping to 0x7FFF if $t2 set
slt     $t2, $v0, $t6           # set $t2 if result smaller than 0xFFFF8000
movn    $v0, $t6, $t2           # negative clipping to 0xFFFF8000 if $t2 set

bne     $a2, $s0, loop          # loop if all output data not computed yet
sh      $v0, -2($a2)            # store the imaginary part

# 4 stall cycles on the 24K/24KE - mispredicted branch
```

5.3.4 DSP ASE Assembly Language Version using Q15 Data Type

Performance: 6*N*K + 15*N + 16 = 20,716 cycles for N = 100, K = 32
Memory use: 35 instructions = 140 bytes

```
# $a0 - pointer to x[]
# $a1 - pointer to h[]
# $a2 - pointer to r[]
# $s0 - N (number of samples)
# $s1 - K (number of filter taps, multiple of 2)

sll     $s0, $s0, 2             # multiply N by sizeof(complex)
add     $s0, $s0, $a2           # compute final address in r[]

addiu   $s1, $s1, -2            # compute K-2
sll     $s1, $s1, 2             # multiply K-2 by sizeof(complex)
sll     $s2, $s1, 8             # align it for use by the MODSUB instruction
ori     $s2, $s2, 8             # insert the MODSUB decrement value (8)
move    $s5, $s1                # save $s1 to use for loop comparison

addiu   $s4, $a1, 4             # offset h[] pointer for LWX
```

```
addiu  $s3, $a0, 4              # offset x[] pointer for LWX

mult   $zero, $zero             # zero accumulator $ac0
mthi   $zero, $ac1              # zero MSBs of accumulator $ac1
mtlo   $zero, $ac1              # zero LSBs of accumulator $ac1

loop:                           # inner loop unrolled 2x

lwx    $t3, $s1($s4)            # load 1st complex coefficient from h[]
lwx    $t2, $s1($s3)            # load 1st complex sample from x[]
lwx    $t1, $s1($a1)            # load 2nd complex coefficient from h[]
lwx    $t0, $s1($a0)            # load 2nd complex sample from x[]

mulsaq_s.w.ph  $ac0, $t2, $t3   # sRe += (a.re * b.re) - (a.im * b.im)
rotr   $t3, $t3, 16             # swap real and imaginary parts
dpaq_s.w.ph    $ac1, $t2, $t3   # sIm += (a.re * b.im) + (a.im * b.re)

modsub $s1, $s1, $s2            # decrement the pointer and wrap around

mulsaq_s.w.ph  $ac0, $t0, $t1   # sRe += (a.re * b.re) - (a.im * b.im)
rotr   $t1, $t1, 16             # swap real and imaginary parts

bne    $s1, $s5, loop           # loop if MODSUB has not wrapped around
dpaq_s.w.ph    $ac1, $t0, $t1   # sIm += (a.re * b.im) + (a.im * b.re)

# 4 stall cycles on the 24K/24KE - mispredicted branch

extr_s.h       $v0, $ac0, SCALE  # scale and saturate the real part
extr_s.h       $v1, $ac1, SCALE  # scale and saturate the imaginary part

addiu  $a0, $a0, 4              # point $a0 to the new start position in x[]
addiu  $s3, $a0, 4              # offset x[] pointer for LWX
addiu  $a2, $a2, 4              # increment r[] pointer by sizeof(complex)

mult   $zero, $zero             # zero accumulator $ac0
mthi   $zero, $ac1              # zero MSBs of accumulator $ac1
mtlo   $zero, $ac1              # zero LSBs of accumulator $ac1

ins    $v0, $v1, 16, 16         # combine the real and imaginary parts

bne    $a2, $s0, loop           # loop if all output data not computed yet
sw     $v0, -4($a2)             # store the result

# 4 stall cycles on the 24K/24KE - mispredicted branch
```

5.4 Example 3: Vector Maximum

The vector maximum algorithm finds the largest element in a vector of input data. It is a non-typical DSP algorithm since it does not utilize MAC operations. The data type is unsigned byte, which is common in image processing and some communications applications.

5.4.1 C version

```
unsigned char x[N];
unsigned char max = 0;
```

```
for (i = 0; i < N; i++)
    if (x[i] > max) max = x[i];
```

5.4.2 MIPS32® Assembly Language Version

Performance: 14*N/4 + 6 = 356 cycles for N = 100
Memory use: 16 instructions = 64 bytes

```
# $a0 - pointer to x[]
# $a1 - N (number of samples, multiple of 4)
# $v0 - result

add    $a2, $a0, $a1             # compute final address in x[]
or     $v0, $zero, $zero         # initialize the max variable

loop:                            # loop unrolled 4x

lbu    $t0, 0($a0)               # load 1st element
lbu    $t1, 1($a0)               # load 2nd element
lbu    $t2, 2($a0)               # load 3rd element
lbu    $t3, 3($a0)               # load 4th element

sltu   $t4, $v0, $t0             # set $t4 if 1st element larger than maximum
movn   $v0, $t0, $t4             # ... and if so, copy it to the maximum

sltu   $t4, $v0, $t1             # set $t4 if 2nd element larger than maximum
movn   $v0, $t1, $t4             # ... and if so, copy it to the maximum

sltu   $t4, $v0, $t2             # set $t4 if 3rd element larger than maximum
movn   $v0, $t2, $t4             # ... and if so, copy it to the maximum

sltu   $t4, $v0, $t3             # set $t4 if 4th element larger than maximum

add    $a0, $a0, 4               # increment x[] pointer by 4*sizeof(char)

bne    $a0, $a2, loop            # loop if all data not processed yet
movn   $v0, $t3, $t4             # ... and if so, copy it to the maximum

# 4 stall cycles on the 24K/24KE - mispredicted branch
```

5.4.3 DSP ASE Assembly Language Version

Performance: N + 17 = 117 cycles for N = 100
Memory use: 19 instructions = 76 bytes

```
# $a0 - pointer to x[]
# $a1 - N (number of samples, multiple of 8)
# $v0 - result

add    $a2, $a0, $a1             # compute final address in x[]
or     $v0, $zero, $zero         # initialize the SIMD max vector

loop:                            # loop unrolled 8x

lw     $t0, 0($a0)               # load 4 elements
lw     $t1, 4($a0)               # load 4 elements
```

```
cmpu.lt.qb     $v0, $t0          # compare to the max vector, set flags
pick.qb        $v0, $t0, $v0     # pick max values based on the flags

addiu  $a0, $a0, 8               # increment x[] pointer by 8*sizeof(char)

cmpu.lt.qb     $v0, $t1          # compare to the max vector, set flags

bne    $a0, $a2, loop            # loop if all data not processed yet
pick.qb        $v0, $t1, $v0     # pick max values based on the flags

# 4 stall cycles on the 24K/24KE - mispredicted branch

precequ.ph.qbl  $v1, $v0         # convert leftmost 2 bytes to a 32-bit word
precequ.ph.qbr  $v0, $v0         # convert rightmost 2 bytes to a 32-bit word

# 1 stall cycle - dependency on $v0

cmp.lt.ph      $v0, $v1          # compare the two max vectors, set flags
pick.ph        $v0, $v1, $v0     # pick the max pair based on the flags

# 1 stall cycle - dependency on $v0

srl    $v1, $v0, 16              # extract the left halfword
ins    $v0, $zero, 16, 16        # extract the right halfword

sltu   $t0, $v0, $v1             # compare maximum values
movn   $v0, $v1, $t0             # select final maximum

srl    $v0, $v0, 7               # compensate the precequ.ph.qb zero padding
```

5.5 Summary

The examples in this chapter illustrate the advantages of using the DSP ASE instruction set for signal processing algorithms. The code becomes faster and often smaller as shown in the coding examples. Note that the performance improvement depends on the algorithm, the data types used, and the optimization effort. Often the speedup is due to the 8-bit or 16-bit data type used by an algorithm, which translates to efficient SIMD processing of four or two elements respectively at a time. Further optimization, e.g., extensive loop unrolling, may further improve the cycle counts at the expense of increased code size, loss of generality, and reduced clarity.

6 MIPS32® DSP ASE Optimization Guidelines

DSP code written using the MIPS32 DSP ASE benefits from:

- General-purpose code optimization techniques applicable to any kind of code and processor
- Optimizations commonly used in highly optimized DSP code, and
- Optimizations specific to the DSP ASE.

This section describes some of these techniques and presents one example from each category.

6.1 Loop Unrolling and Software Pipelining

Loop unrolling is an important code optimization. As illustrated by the examples in this section, the loop management overhead (pointer and index updates, branching to the start of the loop, etc.) can be significant if the loop body is short. By unrolling the loop, this overhead can be amortized over several iterations, thus reducing overall cycle count.

The unrolled loop assumes that the number of iterations of the original loop is multiple of the unroll factor. For example, if a loop is unrolled four times (4x), the number of iterations in the original loop has to be a multiple of four. This is usually not worrisome and the code can be written to work correctly for any number of iterations. Sometimes it is easiest to just produce a few extra don't care values at the tail-end of the unrolled loop; just ensure that the data buffers are large enough to hold all the output values from the unrolled loop.

Another optimization related to loop unrolling is software pipelining. This involves taking a sequence of actions performed sequentially in the body of the original loop in a single iteration, and spreading these actions into stages across several iterations of the loop. This method of software pipelining eliminates data dependency between instructions in a single iteration. This data dependency often causes stalls and reduced performance. The pseudo-code example below illustrates a simple loop with data dependencies and its software-pipelined version.

```
// original loop
for (i = 0; i < N; i++)
{
    a = A(i);
    b = B(a);
    c = C(b);
}
```

```
// software-pipelined loop
a0 = A(0);
b1 = B(a0);
a1 = A(1);

for (i = 2; i < N; i++)
{
    a = A(i);
    b = B(a1);
    c = C(b1);
    a1 = a;
    b1 = b;
}

b = B(a1);
c = C(b1);
c = C(b);
```

In the example on the left, there is a RAW (Read after Write) data dependency after the first two statements in the loop body. If there is a pipeline delay of more than one between those statements, then during the execution of each iteration, the processor will stall after the first and second statements for the required number of cycles.

In the software-pipelined example on the right, the three statements performing the A(), B(), and C() operations are independent. Hence instructions from for example, C(), can be used to fill-in delay slots and cover potential stalls caused by instructions from A() and B(). The order of those calculations can also be exchanged. Also note that the software pipeline has to be warmed up before the loop and drained after the loop. The loop itself iterates N-2 times.

Other minor optimizations include the use of the final buffer address as a loop terminating condition instead of maintaining a separate counter, performing shifts for multiplication and division by powers of two, and identifying common subexpressions and evaluating them only once.

6.2 DSP Code Optimizations

Some optimization methods are seldom used for general-purpose software, but are often seen in DSP code. A typical example is a technique called zipping, which reduces the number of data loads in algorithms like FIR filters. Consider

the calculation of the first two output values of an 8-tap FIR filter. The illustration below shows how the coefficients get multiplied by the data samples:

```
Data samples:  d0  d1  d2  d3  d4  d5  d6  d7  d8  d9
  1st output:  c0  c1  c2  c3  c4  c5  c6  c7
  2nd output:      c0  c1  c2  c3  c4  c5  c6  c7
```

A very naive implementation will load each coefficient and data sample from memory every time they are needed. A more optimized implementation will load the coefficients just once and keep them in registers. It will load data samples d0-d7 first, to compute the first output, and then data samples d1-d8 to compute the second output. With zipping, the optimized implementation will load d0-d8 once and use the loaded values for both output calculations. The relatively large number of general-purpose registers in the MIPS architecture is useful for applying this technique. An even larger number of samples can be kept in registers if the SIMD features of the DSP ASE are used. In this case, another slightly rearranged set of coefficients may be needed, as outlined below for the case of 16-bit coefficients and samples packed into 32-bit words:

```
Data samples:  d0:d1  d2:d3  d4:d5  d6:d7  d8:d9
  1st output:  c0:c1  c2:c3  c4:c5  c6:c7
  2nd output:  00:c0  c1:c2  c3:c4  c5:c6  c7:00
```

Because of the large number of general-purpose registers, especially considering the SIMD features offered by DSP ASE, the MIPS32 cores lend themselves well to algorithms processing more data at a time. For example, it is easy to meet the requirements of a radix-4 FFT implementation, which is faster than a radix-2 implementation but requires a large number of values to be kept in registers during the calculation.

Many algorithms can be implemented in a variety of different ways and often some of these algorithm transformations offer performance advantages. The output results are similar in all cases, but an optimal implementation strikes a good balance between number of registers needed, number of memory operations, number and type of arithmetic operations, regularity of data access patterns, instruction latencies, etc. Make sure the selected algorithm implementation approach is the best match to the architecture.

6.3 Code Optimization Specific to the DSP ASE

Extracting the best performance from the DSP ASE is possible only when the programmer is familiar with all the instructions in the ASE, hence reading the specification as well as the core programming guide is essential.

If the data type is 16-bit or 8-bit, then attempting to rewrite the algorithm in SIMD style using operations directly supported by the DSP ASE instruction set will yield a lower cycle count due to the obtained parallelism. Once the number of instructions required to implement the algorithm are minimized, then they must be scheduled to minimize stalls, taking into account the instruction delay and pipeline interlock conditions. When evaluating the resulting performance, a hand count of the number of cycles must be compared to the cycle count measured on hardware or cycle-accurate simulator to understand the cause of any discrepancies.

As a simple illustration of efficiently using the SIMD capabilities of the processor, consider the task of unpacking YUV image data prior to processing. The YUV color space is commonly used in image and video processing. The color components (U and V) are subsampled with respect to the luminosity (Y) by a factor of two or four in the horizontal and/or vertical dimension. A commonly used format is YUV 4:2:2, which has the color components subsampled horizontally by a factor of two. Hence there is one luminosity value for each pixel, but a UV pair is shared between two adjacent pixels. Video data in YUV 4:2:2 format is commonly transmitted in the following order:

Pixel data: U_0 Y_0 V_0 Y_1 U_2 Y_2 V_2 Y_3 U_4 Y_4 V_4 Y_5 ...

Video processing algorithms usually perform different tasks on each of the Y, U, and V components. In order to use the SIMD capabilities of the 24KE, the pixel data needs to be unpacked, i.e., each of the Y, U, and V values should be separated out and grouped together. Examining the DSP ASE instruction set reveals that some instructions intended for precision reduction and expansion can be used to implement data unpacking:

```
lw          $t0, 0($a0)             # U0:Y0:V0:Y1 - two pixels in YUV 4:2:2
lw          $t1, 4($a0)             # U2:Y2:V2:Y3 - next two pixels in YUV 4:2:2

precequ.ph.qbra     $t2, $t0        # 00:Y0:00:Y1 - half the unpacked Ys
precrq.qb.ph        $t4, $t0, $t1   # U0:V0:U2:V2 - interleaved Us and Vs
precequ.ph.qbra     $t3, $t1        # 00:Y2:00:Y3 - the other half of unpacked Ys
precequ.ph.qbra     $t5, $t4        # 00:V0:00:V2 - unpacked Vs
precequ.ph.qbla     $t5, $t4        # 00:U0:00:U2 - unpacked Us
```

Unpacking the YUV data as illustrated above also has the advantage of converting each data item from 8-bit unsigned to 16-bit unsigned format. This ensures there is enough room for performing the video processing calculations with enhanced precision.

Another example of efficient use of DSP ASE instructions is bitstream unpacking. Many audio and video compression algorithms pack various parameters of different bit widths in a continuous compressed bitstream. The decoder has to first unpack--or extract--the individual values from the bitstream before further processing them. Recognizing the importance of this task, the DSP ASE instruction set includes instructions that facilitate and accelerate bitstream unpacking. One of the accumulator registers is used as a 64-bit data buffer. The EXTP instruction variants extract a specified number of bits and optionally decrement the pos field of the DSPControl register. The pos field holds the number of remaining bits in the bit buffer. The BPOSGE32 instructions checks this number and branches if there are at least 32 bits left. And finally, the MTHLIP instruction is used to reload the bit buffer with a new 32-bit word and at the same time increment the number of available bits by 32. This process is illustrated in the code fragment below:

```
loop:

lbu         $t0, 0($a0)             # size of the data field to extract
addiu       $a0, $a0, 1             # increment size table pointer
extpdpv     $v0, ac0, $t0           # extract a value from ac0, pos -= size
addiu       $a1, $a1, 4             # increment output data pointer
bposge32    loop                    # loop back if pos >= 32
sw          $v0, -4($a1)            # store the extracted value

lw          $t1, 0($a2)             # load next bitstream word
addiu       $a2, $a2, 4             # increment bitstream pointer
bne         $a2, $a3, loop          # loop until no more data available
mthlip      $t1, ac0                # update ac0 bit buffer, pos += 32
```

The illustrated code loads the size (bit width) of each field to be extracted from memory. The performance of the loop can be further improved if the sizes are statically known.

7 Summary

This white paper presents a brief overview of digital signal processing and introduced the DSP application-specific extension to the MIPS32 architecture. Several examples were given illustrating the coding style and performance advantage of the DSP ASE. The guidelines at the end of the white paper summarize some important optimizations to consider when writing DSP software.

For other resources about the cores or the architecture, please refer to the following documents:

- *MIPS32® Architecture for Programmers VolumeIV-e: The MIPS® DSP Application-Specific Extension to the MIPS32 Architecture* (MD00374)
- *Programming the MIPS32® 24KE™ Core Family* (MD00458)