Programming the MIPS® 74K™ Core Family for DSP Applications

This paper introduces programming techniques for the MIPS 74K family cores. In addition to describing the pipeline operation, the paper also provides pipeline timing information that is useful for fine-tuning code execution speed. The paper presents both general-purpose code optimization guidelines as well as advice specific to the 74K family of cores.

Document Number: MD00544
Revision 01.21
May 6, 2009

Contents

1 Overview

The MIPS® 74K™ core family is a high-performance 32-bit RISC core introducing superscalar out-of-order execution to the lineup of efficient RISC cores offered by MIPS Technologies. While substantially different in design, the 74K cores remain fully code compatible with previously released 32-bit MIPS processor cores such as the 4KE™ and 24K™ cores by implementing the same Release 2 of the MIPS32 instruction set. Furthermore, the 74K cores also include Revision 2 of the digital signal processing application-specific extension (DSP ASE) that was introduced in the 24KE™ and 34K™ cores. Revision 2 of the DSP ASE adds new instructions to accelerate video and image processing algorithms and helps improve the performance of signal-processing and multimedia applications.

This white paper introduces the 74K cores from the viewpoint of a software engineer writing and optimizing general-purpose and DSP code. It also presents several software optimization techniques and effective programming guidelines that can help improve performance. The paper is organized as follows:

Section 1 "Overview" is this overview.

Section 2 "The MIPS® 74K™ Processor Cores" introduces the 74K cores. It provides an overview of the most important architectural features and describes the operation of the pipelines. Pipeline timing information, useful for hand-scheduling code to the 74K cores, is also provided. This section also discusses briefly the DSP ASE instruction set.

Section 3 "Software Optimization for the MIPS® 74K™ Core Family" presents software optimization guidelines and methods divided into three categories. The generic methods are applicable to any kind of code running on any processor. The DSP-specific optimizations are related to DSP and multimedia code. And finally, the 74K core-specific optimization guidelines provide advice on improving the performance of code running on the 74K cores.

And finally, Section 4 "Summary" contains a brief summary.

2 The MIPS® 74K™ Processor Cores

Introduced in 2007, the MIPS32® 74K™ processor cores are modern high-performance RISC processors that execute Linux-like operating systems and other control code very efficiently. The dual-issue pipeline is also very efficient for computationally intensive applications in areas such as networking, multimedia, and printing. The 74K core implements MIPS32 R2 architecture and hence provides full binary compatibility with other MIPS32 R2 cores such as the 4KE, 24KE, 34K, etc. At the same time advanced microarchitecture improvements allow them to reach impressive performance levels, which enables new and more demanding applications to achieve real-time performance and fast response times.

This section introduces the main features of the 74K cores. The short overview below is followed by more detailed subsections discussing the pipelines and the DSP ASE instruction set. Knowledge of these two features is important for extracting the best possible performance from the 74K cores.

2.1 Quick Overview

Here is a brief summary of the most important implementation characteristics of the 74K cores. For more detailed information on the 74K implementation, please refer to "Programming the MIPS32 74K Core Family," document

number MD00541. The focus here is on topics relevant to improving the performance of code running on the 74K cores.

- Microarchitecture
    - Limited dual-issue core
      One ALU pipeline and one load/store pipeline. Multiply and divide operations are issued to the ALU pipeline but then continue execution in a dedicated pipeline.
    - Out-of-order integer dispatch unit
      Two eight-entry decode/dispatch queues (DDQs) -- one for each of the two pipelines.
    - Static and dynamic branch and return address prediction
      “Gskew” predictor using three 256-entry tables with two bits per table entry. Different combinations of PC and global history are used to index the tables; majority vote decides the predicted direction.
      8-entry return prediction stack.
    - 16-64KB cache size
      4-way set associative with 32-byte line.
    - Instruction Cache Prefetching
      Configurable option to enable instruction cache prefetching of 0, 1 or 2 next lines.
    - 4KB-1MB scratchpad RAM
      Static RAM with fast access time.
    - On-chip memory management unit (MMU)
      4KB-256MB page size; required by many operating systems.
- Instruction Sets
    - MIPS32 Release 2
      Full compatibility with previous generations of 32-bit processor cores.
    - DSP ASE Revision 2
      SIMD instructions for accelerating DSP and multimedia algorithms.
    - MIPS16e
      Compact 16-bit instruction set useful for reducing code size.
- Optional Units
    - CorExtend interface
      Used to add custom user-defined instructions (UDIs) to the core.
    - IEEE 754 floating point unit
      Implements its own register file for floating-point operands.

It is evident from the feature list above that the 74K cores are modern RISC processors with advanced performance improvement features. All of these features work smoothly together and require no programmer effort to not only guarantee correct execution of any code, but also automatically provide acceleration using the dual-issue out-of-order execution engine along with the caches and the branch predictor. In cases where maximum performance is required, a more detailed understanding of the pipelines helps to schedule the code for optimal execution. Also, significant accel-

eration of DSP and multimedia code can be achieved by using the DSP ASE instruction set, which offers specialized instructions and SIMD processing capabilities. The 74K pipelines and the DSP ASE instruction set are the topics of the next two sections.

2.2 The Pipelines

As mentioned above, the 74K cores are superscalar with limited dual-issue and out-of-order instruction dispatch. There are two main pipelines, each specialized to execute specific instructions:

- AGEN pipeline
  Handles load/store instructions, branches, and conditional move instructions.

- ALU pipeline
  Handles all other instructions including arithmetic, computation, and DSP ASE instructions.

Each instruction is examined and issued to one of the main pipelines depending on the instruction type. Instructions that require the Multiply/divide unit (MDU)-like multiply/divide operations and instructions accessing the accumulators-go first to the ALU pipeline and then branch out to a dedicated MDU pipeline. The two main pipelines help improve performance by executing a load/store operation in parallel with an ALU operation whenever possible.

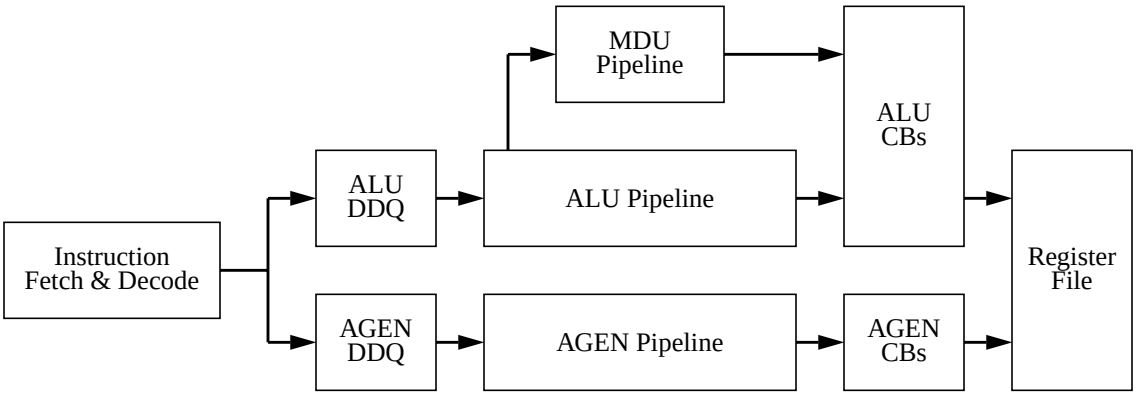

Figure 2-1 Simplified illustration of the 74K core pipeline

After the instruction fetch stages at the beginning of the pipeline, pairs of instructions from the instruction stream are placed into a 2-entry buffer. Each of the two instructions in this buffer is examined and sent to one of the two decode/dispatch queues (DDQs) depending on the instruction type, as shown on Figure 2-1 above. One of the DDQs is associated with the ALU pipeline and receives instructions that will execute on this pipeline. The other DDQ accepts instructions for the AGEN pipeline. Each DDQ can hold up to eight instructions.

The 74K cores can dispatch up to two instructions every clock cycle: one to the ALU pipeline and another to the AGEN pipeline. The operand dependencies of the instructions in both DDQs are examined every clock cycle and one ready to run instruction in each DDQ is selected for execution. Thus, instructions can be executed out of order depending on their operand availability. However, note that loads cannot surpass stores in the AGEN pipeline, i.e., load instructions cannot be executed before store instructions in program order even if the load instructions are ready to execute. This guarantees correct handling of read-after-write memory hazards.

Instructions that go through the ALU and AGEN pipelines generate their results and hold them in temporary storage called ALU/AGEN completion buffers (CBs). The results are moved from the completion buffers to the corresponding architectural state-usually general-purpose registers (GPRs)-when the instruction graduates. Graduation, the last stage in the pipeline, ensures that instructions that were dispatched out-of-order commit their results in program order and hence maintain the correct functionality of the code.

When an instruction in one of the DDQs has a dependency on an instruction that has completed execution but not graduated yet, the data needed by the dependent instruction is forwarded from the appropriate completion buffer. In case one of the instructions in the pipeline generates an exception or a branch turns out to be mispredicted, the data in the CBs for instructions that come after the current one in program order is simply discarded and those instructions do not graduate. This in effect rolls back the processor to the state just before the exception or the branch mispredict. The processor then can take action to handle the exception or follow the correct branch path.

2.2.1 The ALU Pipeline

ALU execution occupies two pipeline stages to allow the high frequency of the synthesizable 74K cores. Although integer instructions typically have result delay of one cycle, some of the most commonly used instructions have been implemented with a result delay of zero cycles. DSP ASE instructions that require saturation have result delays of two cycles. If the result delay requirement between an instruction producing a result and another one using it are not met, the processor will dispatch another ready to execute instruction between the dependent instructions if possible, otherwise the pipeline will stall.

Multiply and divide instructions execute in a separate MDU pipeline and execute there. Other independent ALU instructions can be executed while the multiply/divide unit is busy. There is a dedicated write port on the ALU completion buffer for MDU results, which avoids write port contention issues between the ALU and the MDU.

2.2.2 The MDU Pipeline

In addition to integer multiply and divide operations, the MDU pipeline also executes DSP ASE multiplication-type instructions and instructions that access the accumulators. The result delay of MDU instructions is typically higher than the result latency of ALU instructions because the operations performed by the MDU are more complex. For example, the result delay of a multiply operation that returns the result in a GPR register is six cycles.

At the same time the MDU pipeline is optimized for typical DSP algorithms executing lots of multiply-and-accumulate (MAC) operations. There is internal forwarding path allowing the MDU to execute a sequence of such instructions back-to-back with zero delay.

2.2.3 The AGEN Pipeline

The AGEN pipeline executes the two conditional move instructions MOVN and MOVZ, which take three or more cycles to execute. Since the branch mispredict penalty on the 74K is 12 or more cycles, the conditional move instructions can still be a better choice for unpredictable control-type branches.

Assuming loads hit in the cache, the load to use delay is two cycles for ALU consumers or three cycles if the consumer is another load instruction and the result of the first load is used for address calculation of the second. The core supports up to four outstanding cache misses.

As already mentioned, loads appearing after stores in program order cannot execute before the stores. This fact is sometimes important for performance optimization. Pipeline timing information, which is also important for software optimization, is summarized in the following subsection.

2.2.4 Pipeline Timing

Generating the optimal sequence of instructions to accomplish a given task is the first software optimization step. The next step is scheduling this instruction sequence for the 74K pipelines, which depends on instruction latencies and dependencies in the pipeline.

More specifically, the result delay-the number of cycles needed to compute a result-of each instruction plays an important role in instruction scheduling. Result delay of zero cycles means that the instruction immediately following the one generating the result can use it right away. A non-zero result latency means that a specified number of cycles is needed to compute the result. A subsequent instruction trying to use the result before it is ready will be delayed, i.e., the 74K core will start executing other instructions out of order if such instructions are available and ready to run. This implies that in many cases the result delay will be hidden by the 74K core and will not affect negatively the overall cycle count. In other cases the delay may be visible providing an opportunity for further optimization.

The following table summarizes the result delays and other timing information of the 74K cores. The notes provided after the table provide explanation. An introduction to DSP ASE along with some additional details is given in the next subsection.

Table 2.2 Pipeline timing information for the 74K™ cores

Operations	Delay	Notes
Logical operations: AND, ANDI, NOR, OR, ORI, XOR, XORI	0	(1)
Aritmetic operations: LUI, ADD (rt = 0), ADDU (rt = 0)	0	(1), (2)
Arithmetic operations: ADD, ADDU, ADDI, ADDIU whose destination register consumers are in ALU pipe	0	(2)
Shift operations: SLL (shift <= 8), SRL (31 <= shift <= 25)	0	(1)
Condition testing: SLT, SLTI, SLTIU, SLTU	0	(1)
Conditional move MOVN / MOVZ	3+	(3)
Other MIPS32 ALU operations	1	(4)
DSP ASE non-saturating ALU operations	1	(5)
DSP ASE saturating ALU operations	2	(5)
GPR-targeting multiply operations	6 / 7	(6)
MAC to MAC operations	0	(7)
MAC_SA to MAC or MAC_SA operations	3	(8)
MAC to EXTR oprations	4	(9)
MAC to MFLO / MFHI operations	4	(9)

Table 2.2 Pipeline timing information for the 74K™ cores

Operations	Delay	Notes
Accumulator read-to-use	6	(10)
Load-to-use delay	2 / 3	(11)
Address setup for loads and stores	2	(12)
Branch mispredict penalty	12+	(13)
Instruction fetch address change penalty	3	(14)

Notes:

(1) Frequently used ALU operations can execute back-to-back without stalls.

(2) The special case of an ADD or ADDU instruction with register r0 as a second input register is often used by compilers to synthesize register move operations. Such instructions have zero cycles result delay. Also, the following integer operations ADD, ADDI, ADDU, ADDIU can generate the result and bypass back to the ALU pipe in 1 cycle, thus allowing for back to back execution of these instructions.

(3) MOVN / MOVZ delay is at least three cycles because the instruction is issued twice. Also, note that these instructions are executed in the AGEN pipeline.

(4) The remaining MIPS32 ALU instructions have a result delay of 1 cycle. Back-to-back dependent instructions will stall for 1 cycles if there are no other instructions to execute out-of-order.

(5) Saturating ALU instructions have longer result delay than non-saturating ALU instructions. For example, ADDQ.PH has a result delay of 1 cycle like standard ALU instructions, but ADDQ_S.PH has a result delay of 2 cycles.

(6) Delay is 7 cycles if the result is used for address generation in a following load/store instruction. In all other cases the result delay is 6 cycles.

(7) A sequence of MAC instructions will execute without any delays even though there is dependency on the accumulator. For example:

```
DPAQ_S.W.PH    ac0, $t0, $t1
DPAQ_S.W.PH    ac0, $t2, $t3
```

(8) MAC instructions requiring saturation of the accumulator (the "_SA" option after the root mnemonic) need the accumulator for a few extra cycles. Hence, executing a sequence of such instructions, for example, DPAQ_SA, will incur a delay of 3 cycles between each pair.

(9) Accumulator access operations like EXTR, MFLO, and MFHI are usually used at the end of a computation to retrieve the final result from the accumulator into a GPR register. Hence even if the result delay between the final MAC inside the processing loop and an EXTR instruction outside the loop cannot be hidden, the performance impact may not be significant.

(10) This is the delay between an instruction transferring data from an accumulator to a GPR register (e.g., EXTR, MFLO, and MFHI) and an instruction using this GPR register. Again, a sequence like that is usually used at the end of a processing loop and the impact of the large result delay is not significant.

(11) Load-to-use delay is 2 cycles if the consumer is an ALU pipeline operation, or 3 cycles if the consumer is an AGEN pipeline operations. Exceptions include:

- Delay is 2 cycles for store data.
- Delay is 2 cycles if the AGEN pipeline operation is a branch.

(12) Delay from modifying a register to using it for address generation. For example, there will be two delay cycles between the following two instructions:

```
ADDIU   $a0, $a0, 16
LW      $t0, 0($a0)
```

(13) Branch mispredict penalty is minimum 12 cycles. Branch likely mispredicts have higher mispredict penalty in that the branch likely instruction (and the delay slot if taken) itself is re-executed.

(14) Unconditional as well as correctly predicted taken branches stall instruction fetch for 3 cycles because the address used to fetch instructions from changes. However, instruction dispatch and execution continue since there are typically some instructions in the DDQs at the time the branch is taken.

The instruction timing information presented here should be used to create code that is better suited for optimal execution on the 74K cores. More software optimization guidelines are provided in the last section of this white paper, but before moving to that topic let's first introduce the DSP ASE instruction set that enhances the 74K cores with instructions specifically crafted for multimedia applications.

2.3 The MIPS32® DSP ASE Instruction Set

The 74K cores implement Revision 2 of the DSP ASE instruction set intended to accelerate algorithms in areas such as audio/video compression, image processing, communications, etc. There are two main advantages of using the DSP ASE instruction set. First, many of the DSP ASE instructions work in SIMD mode and perform a specified operation simultaneously on several data elements packed into 32-bit vectors. And second, DSP ASE instructions automatically perform additional operations such as rounding and saturation that otherwise require a significant number of processor cycles to implement.

This section assumes that the reader is familiar with the basic digital signal processing concepts-and more specifically, the operations and data types typically used in signal processing algorithms.

2.3.1 The DSP ASE Features

The DSP ASE enhances the MIPS32 Release 2 architecture with a set of new instructions improving the performance of signal processing and multimedia applications. It uses the single instruction multiple data (SIMD) approach to process data packed into 32-bit registers as vectors of smaller-sized data and supports data types and operations typically used in multimedia-oriented and general-purpose DSP algorithms. For example, four 8-bit or two 16-bit values can be packed into a single 32-bit register.

The DSP ASE also adds new state to the basic MIPS32 Release 2 architecture. 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, containing several status flags as well as few data fields used by some of the instructions, has also been added.

The speedup obtained using the DSP ASE compared to a purely MIPS32 implementation depends on the data types used, the type and frequency of DSP instructions in the code, the algorithm, and the pipeline characteristics.

Note that the DSP ASE does not turn the 74K cores into a high-end specialized digital signal processor. Such processors cost significantly more than the 74K cores and are typically less suited to run operating systems such as Linux and general-purpose (i.e., non-DSP) code efficiently. Instead, the main goal of the DSP ASE instruction set is to give a boost to the DSP performance of the MIPS32 cores without significant clock speed degradation or area increase. This is accomplished by utilizing clever design techniques that reuse the hardware that is already present in the core. Thus, the DSP ASE is a cost-efficient way to enhance performance, reduce power consumption, and integrate more signal-processing functionality into user applications.

The following short list is an overview of the most important features offered by the DSP ASE:

- Efficient SIMD data types and operations
    - Two 16-bit elements packed into one 32-bit register
    - Four 8-bit elements packed into one 32-bit register
- Variety of supported data types
    - Signed fractional: 16-bit (Q15) and 32-bit (Q31)
    - Unsigned integer: 8-bit
- Four 64-bit accumulators
- Single-cycle repeat rate for most instructions
- Specialized instructions spawning multiple categories:
    - SIMD arithmetic operations
    - SIMD multiply-accumulate (MAC) and regular multiply operations
    - SIMD precision expansion and reduction operations
    - SIMD arithmetic and logical shift operations
    - SIMD compare and pick operations
    - Accumulator result extract operations
    - Bit manipulation and bit field extract operations
    - Indexed load operations
- Rounding and saturation options for many instructions
- DSPControl register holding various flags and fields, including:
    - Condition code bits used for compare-and-pick operations

- Overflow/underflow flags
- Carry bit for implementing long additions
- Position and size used by bit-field insert/extract instructions

The next subsection presents more details about the DSP ASE instruction set.

2.3.2 The DSP ASE Instructions

Although a full description of the DSP ASE instruction set is beyond the scope of this paper, this subsection introduces the instruction naming conventions and presents several examples illustrating some of the most frequently used instructions.

Each DSP ASE instruction has several variants that work with different data types and in some cases include rounding and saturation of the result. The desired data types and operation modes are encoded in the instruction mnemonic as described below:

- "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.
- "H" suffix in the root mnemonic
  Specifies that the results are halved (divided by two) before writing them to the destination register.
- "_R" suffix: rounding mode
- "_S" suffix: saturation mode
- "_RS" suffix: both rounding and saturation
- "_SA" suffix: 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
- "L" or "R" after the type selector: access the left or the right half of the source register

- "LA" or "RA" after the type selector: access alternating left-aligned or right-aligned values

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 only a subset of all options described above is available for each instruction. Please consult the "DSP ASE Architecture for Programmers" Reference Manual (MD00374) for information about the instruction variants supported by the DSP ASE. The DSP ASE instructions are also described in Chapter 7 of the "Programming the MIPS32 74K Core Family" Manual (MD00541).

The rest of this subsection presents examples of DSP ASE instructions to introduce the flavor of this instruction set. The instructions are available both with and without the options enclosed in square brackets; their operands are indicated at the rightmost side of the line containing the instruction name. The list is by no way comprehensive. The reader is encouraged to refer to the DSP ASE reference manual mentioned above for more information.

- ADDQ[_S].PH and SUBQ[_S].PH rd, rs, rt
  Compute the two 16-bit sums or differences of the corresponding 16-bit fractional values in each of the source registers rs and rt. The two results are packed and written to the destination register rd after optional saturation to the range [-1.0; +1.0), which for 16-bit fractional values corresponds to the integer range [-32768; +32767].

- ADDUH[_R].QB and SUBUH[_R].QB rd, rs, rt
  Compute the four 8-bit sums or differences of the corresponding 8-bit unsigned values in the source registers rs and rt. Unlike ADDQ and SUBQ, the results are computed internally with 9-bit precision and are then halved by shifting them right by 1 bit. This avoids any possibility of result overflow and scales the results by , which is convenient for some algorithms. There is optional rounding, which adds 1 to the intermediate 9-bit results before the right shifts.

- ABSQ_S.QB, ABSQ_S.PH, and ABSQ_S.W rd, rs
  Compute the absolute value of each 8-, 16-, or 32-bit element of the source register rs and place the results in the destination register rd. The results are saturated, i.e., taking the absolute value of -1.0 is handled appropriately.

- RADDU.W.QB rd, rs
  Unsigned reduction add instruction that computes the sum of all four unsigned 8-bit elements of the source register rs and places the 32-bit sum in the destination register.

- PRECRQ.QB.PH and PRECR.QB.PH rd, rs, rt
  The first instruction reduces the precision of four 16-bit fractional values taken from the two source registers to eight bits by dropping the eight LSBs from each 16-bit value. It can also be used to pack odd-numbered bytes from two source registers into one register. Similarly, the second instruction reduces the precision of four 16-bit integer values and can be used to pack even-numbered bytes from two source registers into one register.

- MULQ_RS.PH and MUL[_S].PH rd, rs, rt
  As already mentioned, the first instruction computes two 16-bit fractional multiplications of the corresponding elements in the two source registers and writes the packed results to rd after rounding and saturation. The second instruction operates similarly, but performs integer multiplications.

- DPAQ_S.W.PH and DPA.W.PH ac, rs, rt
  Perform dot product accumulate (two MAC operations) by first computing two full-precision 32-bit products of the corresponding 16-bit elements of the two source registers and then accumulating the two products to one of

the four 64-bit accumulators ac. Can also be used to compute the imaginary part of a sequence of complex MAC operations. The first instruction works with fractional data, while the second one works with integer data.

- DPAQX_S.W.PH and DPAX.W.PH ac, rs, rt
  Like above, but compute cross-products, i.e., multiply the 16-bit data elements in the left and right parts of rs by the 16-bit data elements in the right and left parts of rt.

- MULSAQ_S.W.PH and MULSA.W.PH ac, rs, rt
  Compute the real part of a sequence of complex MAC operations. First, calculate two full-precision 32-bit products of the corresponding 16-bit elements of the two source registers. Then subtract the two products, and finally accumulate the difference to one of the four accumulators.

- CMPU.EQ.QB, CMPU.LT.QB, and CMPU.LE.QB rs, rt
  Compare the corresponding unsigned 8-bit elements from the two source registers for equality (EQ), less-than (LT), or less-than-or-equal (LE) and record the result of the comparison in the four ccond bits of the DSPControl register. The PICK.QB instruction described below uses these bits to selectively copy one of the corresponding 8-bit elements to the destination register.

- PICK.QB rd, rs, rt
  Used in conjunction with one of the CMPU instructions above, this instruction selects the appropriate elements from rs and rt based on the ccond bits in the DSPControl register. Each 8-bit element is copied from rs, if the corresponding ccond bit is one, or from rt, if the ccond bit is zero. The combination of CMPU and PICK allows four compare-and-pick operations to be performed by just two instructions.

- EXTR_RS.W rd, ac, shift
  Extract a 32-bit value from one of the accumulators into the destination register rd. A shift right by a specified number of bits is performed prior to the extraction and the result is rounded and saturated. This instruction is typically used at the end of a computational loop to obtain and scale the final result.

- EXTPDP rd, ac, size
  Extract a specified number of bits (size) from a bit buffer maintained in one of the accumulators. The pos field of the DSPControl register determines the extraction start position. It is automatically decremented by size, making it ready for subsequent use of the same instruction. Other instructions check the number of remaining bits in the bit buffer and, if necessary, load new data into it.

- LBUX, LHX, and LWX rd, offset(base)
  Indexed load of an unsigned byte, a 16-bit halfword, or a 32-bit word from address base+offset. Base and offset are both registers. These instructions decrease pointer maintenance overhead by allowing the programmer to access several data arrays at different base addresses but modify only one register holding the offset.

It should be evident from the list of selected DSP ASE instructions above that DSP ASE is specialized but versatile enough to target a wide range of applications with different computational and precision requirements. The amount of acceleration provided by DSP ASE depends on the data types used and the instruction mix. Image and video processing, where the 8-bit data type is commonly used, has the biggest potential for acceleration since the 4-way SIMD instructions working with this data type can process four data elements simultaneously.

Now that we have introduced the features of the MIPS 74K cores, it is time to discuss software optimization for the 74K pipelines. As previously noted, even though the 74K cores automatically improve performance by out-of-order instruction execution on dual pipelines, optimizing the code and scheduling it preoperly is still important.

3 Software Optimization for the MIPS® 74K™ Core Family

The final section of this paper deals with software optimization. This is a broad topic that cannot be compressed into few pages of text. Many of the common optimization techniques are applied directly by the compiler and may need no special action on the part of the programmer. This section therefore presents some generic software optimization guidelines applicable to any processor but also provides specific advice for improving the execution performance of code scheduled for the 74K cores.

3.1 Generic Software Optimization Guidelines

There are plenty of performance improvement tricks that can be applied to variety of software running on different processors. The C compilers for these processors can usually carry on such optimizations automatically, which is very handy considering that the majority of code is written in C today. However, a small fraction of performance-critical code is still written in assembly language in order to extract the best possible performance out of the processor. Hence, it is worthwhile to examine briefly some of the generic software optimization techniques.

3.1.1 Strength Reduction and Invariant Code Motion

Imagine an expression depending on a loop variable. Since the loop variable is changing systematically (at least for simple loops), rather than evaluating the expression every time, its computation can be simplified by reusing the value of the same expression from the previous loop iteration. Moreover, if part of the expression does not depend on the loop variable, its calculation can be moved outside the loop. Here is a simple example in C illustrating these ideas:

```
// original code
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        frame_buffer[(y0 + y) * FRAME_WIDTH + x0 + x] = image[y][x];
```

```
// same code with strength reduction
frame_row = &frame_buffer[y0 * FRAME_WIDTH];
for (y = 0; y < image_height; y++)
{
    for (x = 0; x < image_width; x++)
        frame_row[x] = image[y][x];
    frame_row += FRAME_WIDTH;
}
```

Note how the computationally expensive expression used to index into the frame_buffer array has been eliminated. The frame_row pointer is set to the beginning of row y0 outside of the loops and then updated in the outer loop to point to the beginning of each subsequent row.

As already mentioned, compilers typically perform such optimizations automatically.

3.1.2 Inner/Outer Loop Interchange

This optimization is applicable for nested loops and refers to exchanging the inner loop with the outer loop. The intent is to make the memory accesses more straightforward and improve the locality of reference. The following example illustrates this simple optimization:

```
// original code
for (x = 0; x < width; x++)
    for (y = 0; y < height; y++)
        image[y][x] = 0;



// same code with loop interchange
for (y = 0; y < height; y++)
    for (x = 0; x < width; x++)
        image[y][x] = 0;
```

Although almost identical, the second version of the code will have much better performance because it accesses contiguous memory locations as the innermost loop variable x increments. Note that per C convention, 2D arrays are stored row-first in memory. Hence all data elements within a row are contiguous. This simplifies address generation, but most importantly, takes advantage of the processor's data cache locality.

3.1.3 Loop Unrolling And Software Pipelining

Loop unrolling is an important code optimization method. As illustrated by the examples in this subsection, 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.

Loop unrolling assumes that the number of iterations of the original loop is a 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 limitation is usually not a problem 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 output values.

Another optimization related to loop unrolling is software pipelining. It 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. Software pipelining eliminates data dependencies between the instructions in a single iteration. These data dependencies often cause pipeline delays and reduce 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 data dependency between the first and the second statements in the loop body, as well as between the second and the third statements. If the instructions corresponding to the first and the second statements, i.e., A(i) and B(a), have non-zero result delays, then the processor may have to stall the pipeline waiting for the results to become available. The 74K core will most likely find suitable instructions to execute and avoid the stall, but it is best to guarantee that the result delay requirements of all instructions are met and hence the processor will never have to stall.

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.

This concludes our brief coverage of generic optimization techniques. Other small 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.

3.2 DSP Code Optimization

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:

```
 Input data samples:       d0  d1  d2  d3  d4  d5  d6  d7  d8  ...
Coefficients for y0:       c0  c1  c2  c3  c4  c5  c6  c7
Coefficients for y1:           c0  c1  c2  c3  c4  c5  c6  c7

 First output (y0):    c0d0 + c1d1 + c2d2 + c3d3 + c4d4 + c5d5 + c6d6 + c7d7
Second output (y1):    c0d1 + c1d2 + c2d3 + c3d4 + c4d5 + c5d6 + c6d7 + c7d8
```

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, an even more 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 illustrated below for the case of 16-bit coefficients and samples packed into 32-bit words:

```
 Input data samples:       d0:d1  d2:d3  d4:d5  d6:d7  d8:d9
Coefficients for y0:       c0:c1  c2:c3  c4:c5  c6:c7
Coefficients for y1:       00:c0  c1:c2  c3:c4  c5:c6  c7:00
```

The first set of coefficients is used to compute y0 and all even-numbered output samples. The second set is used for y1 and all odd-numbered output samples.

Because of the large number of general-purpose registers, especially considering the SIMD features offered by DSP ASE, the 74K cores lend themselves well to algorithms processing more data elements 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 needs to keep a large number of values 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 the best balance between number of registers needed, number of memory operations, number and type of arithmetic operations, regularity of data access patterns, result delays, etc. Make sure the selected algorithm implementation approach is the best match to the architecture.

It should be noted that a reasonable degree of familiarity with the DSP instructions will allow the programmer to extract the best performance. The architecture specification and the core programming guide provide the necessary information.

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 is minimized, the instructions must be scheduled taking into account their result delays. When evaluating the resulting performance, obtaining a trace from one execution will illuminate the cause of stalls and also facilitate optimization.

3.2.1 Pixel Unpacking Example

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:      U0  Y0  V0  Y1  U2  Y2  V2  Y3  U4  Y4  V4  Y5  ...
```

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 74K core, 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

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 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.

Note that the example above as well as the following examples have not been scheduled to the 74K core pipelines. In a real application other instructions surrounding the illustrated code fragments can be used to fill-in the delays caused by result dependencies.

3.2.2 Sum of Absolute Differences Example

Another interesting DSP ASE example shows the kernel performing the sum of absolute differences (SAD) function used in motion estimation algorithms for video compression. The function accumulates the absolute difference between the pixels from a reference 8x8-pixel block and those from a similar block inside the current video frame. Using DSP ASE, here is how the SAD of four pixels at a time can be calculated and accumulated:

```
SUBUH_R.QB      $t0, $s0, $s1       # subtract 4 pixels with halving
ABSQ_S.QB       $t0, $t0            # find the 4 absolute values
RADDU.W.QB      $t0, $t0            # sum the absolute values
ADDU            $v0, $v0, $t0       # accumulate the result
```

The above sequence is four instructions long and will execute in four cycles when properly scheduled. Performing the same calculation using the MIPS32 Release 2 instruction set will require approximately 20 instructions. The performance advantages offered by the DSP ASE are obvious.

3.2.3 Bitstream Unpacking Example

The last example of efficient use of DSP ASE instructions presented here 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
extpdpv     $v0, ac0, $t0    # extract a value from ac0, pos -= size
addiu       $a1, $a1, 4      # increment output data pointer
addiu       $a0, $a0, 1      # increment size table 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 static and known in advance.

3.3 Optimizing for the 74K™ Core Pipelines

Optimizing for a dual-issue out-of-order superscalar processor is challenging. First, the processor already does a very good job in finding ready-to-execute instructions and using them to cover result delays, memory latencies, and other events that would otherwise cause pipeline stalls. And second, the out-of-order dual-issue execution of the instruction stream makes it more difficult to analyze and optimize the application.

Hence, this subsection presents advice on good coding practices for use with the 74K cores. Following this advice does not provide a 100% guarantee that the code will run in the most efficient way possible, but is a very good start in this direction. It also represents information useful to an optimizing compiler for the 74K pipeline.

1. Use loop unrolling and software pipelining.

The 74K cores work best when there are plenty of instructions to execute with little dependencies between them. Loop unrolling and software pipelining help with this and at the same time reduce the number of overhead instructions - branches, pointer increment, etc.

2. Schedule the code for the 74K pipelines.

Although in many cases the 74K cores can find useful instructions to execute out-of-order and thus cover possible pipeline stalls, it is beneficial to schedule the instructions in a way that takes into account the result delays of all instructions and avoids stall conditions altogether. Note that some common ALU operations have result delay of 1 cycle and the load-to-use delay is 2 cycles. Consult the Pipeline Timing section for more information.

The following example code will execute with two stall cycles because of the data dependency through register $v0:

Expression to compute: ($v0 = $t0 - $t1 - $t2 - $t3)

```
SUBU   $v0, $t0, $t1
SUBU   $v0, $v0, $t2       # 1 stall cycle - $v0 register dependency
SUBU   $v0, $v0, $t3       # 1 stall cycle - $v0 register dependency
```

The code can be rearranged to avoid all of the stall cycles:

Expression can be rewritten as: ($v0 = ($t0 - $t1) - ($t2 + $t3))

```
SUBU   $v0, $t0, $t1
ADDU   $v1, $t2, $t3
SUBU   $v0, $v0, $v1       # 0 stall cycles - ADDU consumer is in ALU pipe
```

Of course, this is an isolated example. In reality it is very likely that there will be other instructions around this code fragment that the 74K core will be able to execute out of order and thus eliminate the stall cycles.

3. Consider the instruction types (ALU / AGEN) when scheduling code.

Multiply instructions returning result in a GPR register have a 6-cycle result delay. However, note that this means six ALU pipe cycles, which is not equivalent to six instructions. For example, the code may have four ALU instructions and two AGEN instructions between the multiply instruction and the consumer of its result, but the two AGEN instructions will most likely execute in parallel with the other ALU instructions. The 6-instruction code fragment will execute in four cycles and the consumer of the multiply result will have to wait two extra cycles.

In some cases replacing a multiply by constant with shifts and adds can help reduce the result delay. For example, multiplication by 5 = 4 + 1 can be accomplished by shifting the multiplicand left by 2 bits (to multiply it by 4) and adding the multiplicand to the result. Compilers often do tricks like that to improve execution speed.

4. Schedule for the longest dependency chain first.

Not all calculations are created equal - some require more instructions (and cycles) than others. Find the longest dependency chain-from the loads through the calculations and finally to the stores-and schedule this chain first leaving blanks where there would be stalls. Then fill-in the blanks with other instructions and keep in mind that some of them may execute in parallel.

5. Schedule the prevailing type of instructions first.

Usually there are more ALU instructions than AGEN instructions (loads, stores, and branches). It makes sense to first schedule the ALU instructions and then insert the AGEN instructions such that they are executed in parallel with the ALU instructions "for free," i.e., not requiring additional cycles.

6. Provide even mixture of ALU and AGEN instructions.

Long sequences of instructions destined for one of the pipelines (ALU or AGEN) can, under some circumstances, drain the DDQ of the other pipeline and at the same time temporarily prevent instructions to be issued to the drained pipeline. For example, if a store instruction in the AGEN DDQ is waiting for a multiplier result, any loads following this store cannot proceed. This means that the AGEN DDQ quickly fills up. The 2-entry instruction buffer sitting between instruction fetch and the DDQs can also fill up with AGEN instructions. At this point no instruction can pass to the ALU pipeline through the 2-entry instruction buffer and the ALU pipeline will eventually be drained. Instructions will start moving as usual once the store is executed. To remedy the situation, mix ALU and AGEN instructions; this will keep the ALU pipeline busy while the store instruction in the AGEN pipe is being processed. Consult the next guideline for an example illustrating this issue.

7. Move stores away from computation especially if followed by loads.

This relates to the previous point. Since loads cannot be executed before stores, make sure you have enough cycles between the instruction producing the result and the store instruction so that the store instruction can execute immediately. Here is an example that also illustrates an issue discussed in the previous guideline:

```
MUL    $v0, $t0, $t1
SW     $v0, 0($a0)

LW     $t0,  0($a1)
LW     $t1,  4($a1)
LW     $t2,  8($a1)
LW     $t3, 12($a1)
LW     $t4, 16($a1)
LW     $t5, 20($a1)
LW     $t6, 24($a1)
LW     $t7, 28($a1)
ADDIU  $a0, $a0, 4
```

Assume that the DDQs are empty at the start of this example. The MUL and the SW will issue simultaneously to the ALU and AGEN DDQ respectively. The following six LW instructions will go through the 2-entry buffer and make it to the AGEN DDQ in three cycles. The AGEN DDQ will now hold a total of seven instructions (1 SW and 6 LW). Note that since loads cannot surpass stores, they cannot start executing. The store depends on the multiplication result which will be ready six cycles after the MUL instruction was issued.

Meanwhile, the last pair of LW instructions will get into the 2-entry instruction buffer, but only one of the LW instructions will be transferred to the AGEN DDQ since it can only hold eight instructions. Since the 2-entry buffer is not empty, the following instructions (the ADDIU in this example) cannot proceed even though the ALU DDQ is empty and the ALU pipeline is idle.

8. Place loads in front of stores whenever possible.

Again related to the same issue. This is accomplished by software pipelining - data for the next iteration of a loop is loaded while the current iteration has not finished yet. The extra load instructions in front of the stores will also give

time for the results needed by the store instructions to compute. In general, if you know that the load and store addresses are going to be different then it is preferable to place the loads ahead of stores.

9. Reduce the number of mispredicted branches using conditional moves.

Mispredicted branches are expensive on the 74K cores: 13+ cycles. Try to reduce their number by further unrolling the loops and using conditional move instructions and other tricks where possible. Here is an example that computes the absolute value of a register:

```
    BGTZ   $v0, positive_value
    NEGU   $v0, $v0
positive_value:
    ...
```

Although the above code fragment is only two instructions long, assuming equal probabilities of encountering positive and negative values-meaning that branch prediction will not be very helpful-the code will execute in approximately eight cycles on average. Here is an alternative:

```
    SLT    $t0, $v0, $zero
    NEGU   $v1, $v0
    MOVN   $v0, $v1, $t0       # 1 stall cycle - $v1 register dependency
```

This variant is three instructions long and will execute in about six cycles since conditional moves take at least three cycles on the 74K cores. Another option is to use an arithmetic trick:

```
    SRA    $t0, $v0, 31
    XOR    $v0, $v0, $t0       # 1 cycle stall - $t0 register dependency
    SUBU   $v0, $v0, $t0
```

This version of the code will execute in five cycles taking into account the two stall cycles illustrated above. The code can be effectively executed in three cycles if the 74K core can find two instructions to fill-in the result delay slots. And finally, here is the last version of the code:

```
    ABSQ_S.W  $v0, $v0
```

This version uses a single DSP ASE instruction. The result delay is two cycles but usually something useful can be done in those two cycles.

10. Do not use very short loops.

The instruction fetch stage encounters a 3-cycle stall even on correctly predicted branches. This situation is normally not visible since the DDQs have enough instructions buffered to cover for it and then instruction fetch quickly catches up. However, on very short loops the effect may be visible.

11. Use data prefetching to improve the cache efficiency.

Several types of prefetch instructions are available on the 74K cores. The simplest ones just alert the 74K core that a specified address will be accessed in the near future. The processor can then fetch the corresponding cache line into the cache so that there is no cache miss when the data is accessed. More specialized variants tell the processor that the data will be accessed only once (as a stream) and need not be retained in the cache or, alternatively, that multiple

accesses to the same data are expected and the data should be retained in the cache. There is even a prefetch hint telling the processor to prepare for store to a specified memory location.

Note that, in order to be effective, prefetch instructions should be placed well ahead of the actual memory access instructions. Also, since the cache always transfers complete cache lines, there is no point to issue more than prefetch instruction to the same cache line.

4 Summary

This paper presented an overview of the MIPS 74K out-of-order superscalar cores from the viewpoint of a software engineer optimizing multimedia and DSP code. The advanced set of architectural features, including the DSP ASE instruction set, make the 74K cores a high-performance modern RISC processor with substantial multimedia processing capacity.