RISC PROCESSOR BASED SPEECH CODEC IMPLEMTATION FOR EMERGING MOBILE MULTIMEDIA MESSAGING SOLUTIONS

Manish Arora, Suresh Babu P. V. and Vinay M. K

Emuzed India Private Limited, #839, 7th Main, 2nd Cross, HAL 2nd Stage, Bangalore, India-560008
Phone: +91-80-5252223/4, Email: arora, pvsuresh, vinaymk@emuzed.com (www.emuzed.com)

ABSTRACT

Mobile Multimedia Messaging (MMS) promises to provide a richer and versatile experience to the user along with new revenue streams for mobile service operators. MMS allows a full content range including images, audio, video and text in any combination. It delivers a location independent, total communication experience to the mobile customer. As proved by the success of Short Message Service (SMS) in generating revenue MMS applications would be the essential drivers of continuous growth in new services beyond voice. Current programmable mobile handsets could suffice as platforms for simple MMS service introduction if the existing capability of the RISC processor of the handset is exercised fully. Speech, an important content in MMS has traditionally been encoded and decoded on DSP processors. This paper describes the challenges and techniques of implementing speech codecs on RISC processors. The specific speech codec implemented was GSM-AMR and the processor used was ARM9TDMI. The techniques described are generic and applicable to any speech codec and RISC processor platform.

1. INTRODUCTION

The global mobile communications industry is currently evolving from voice driven communication to personal multimedia communication. The mobile data services industry has seen a tremendous growth and SMS services have been the first to have financial impact on the operator's revenue. It is predicted that messaging will lead the way to profitability in 3G as well [1]. The extension of SMS, MMS requires high-speed networks for transmitting messages that consist of much more data than the current messages. General Packet Radio Service (GPRS) creates an ideal platform for mobile data applications and services. Since MMS is based on Wireless Application Protocol (WAP), the protocol being bearer and network independent, it enables multimedia messages from a GSM/GPRS network to be sent to a TDMA or WCDMA network [1].

The evolution of messaging content and user benefits may take place as shown in figure 1. From the existing text messaging, the next step would be simple picture messaging. This would be further enhanced with the facility to personalize and create content in the form of graphics and eventually images with digital camera and mobile phone interfacing. The picture would be annotated with text or speech. The final step may be new multimedia content such as audio and video clips [1].

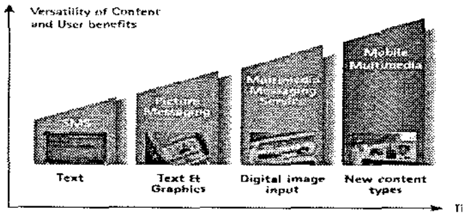

Figure 1. Messaging Content Evolution with Time.

These applications would necessitate improvements in mobile handheld devices and communication protocols. Compression of next generation messaging content would be of prime importance due to bandwidth and storage constraints. To enable fast transfer of multimedia content, they need to be subjected to suitable compression stages. These compression algorithms would be implemented on target handheld devices. Ideal candidates for the implementation of these image and speech compression algorithms for the MMS application would be DSP processors. But the existing programmable mobile devices combining personal digital assistant and phones have limited programmable processing power in the form of RISC processors only. The hardware available is enough for simple applications such as Internet browsing and word processing, along with running the operating system and system functions [2]. A possibility of implementing basic MMS on existing devices lies in efficiently exploiting the processing power of the RISC processor. The need for efficiency in implementation is critical due to limited processing power and direct influence of the firmware on the battery life of the device.

In this paper we describe methods for implementing typical speech codecs on RISC processors for MMS solutions. Section 2 explains typical RISC processor hardware design advantages. Speech codecs and issues with implementing them on RISC architectures are discussed in section 3. Section 4 suggests techniques for efficient processor implementations. Section 5 provides the results and conclusions of our target implementation followed by the references.

2. RISC ARCHITECTURE ADVANTAGES

With the Internet and wireless services coming together digital handheld communication devices are expanding rapidly in capability and hence complexity. The features present in RISC architectures provide distinct advantages in the design, development and product capability for this expanding market. The emphasis of the RISC system design approach is software while maintaining simplicity in the hardware and high compiler efficiency has been an important RISC hardware design goal. This makes the software design process very simple reducing software development time considerably for time and power critical applications. The processor has a single clock and a few very simple instructions. LOAD and STORE are incorporated as different instructions and the rest of the instruction set is register to register. Instruction set orthogonality is easily achievable for RISC instruction sets. The disadvantage is the resulting large code size. The hardware complexity of such designs is low so more transistors can be spent on general-purpose registers. This further improves the overall compiler efficiency. A variety of efficiently silicon sized, low power RISC cores are now available for use in handheld devices powering applications like consumer entertainment, digital imaging, networking, security and wireless devices etc. As an example to our discussion on RISC processor, we discuss the very widely used ARM9TDMI.

2.1. ARM9TDMI

ARM9TDMI is a member of ARM family of 32-bit general-purpose RISC microprocessors, targeted at embedded applications where high performance, low die size and low power are all important. The ARM9TDMI provides a high instruction throughput and impressive real-time interrupt response at a low small die size and cost. The ARM9TDMI supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing user to trade off between high performance and high code density [3]. The Thumb instruction set contains commonly used instructions at 16 bit sizes. Thumb instruction set can be utilized at code size critical portions and complicated DSP portions can be coded using the 32-bit ARM instruction set. In contrast to previous processors such as ARM7TDMI from ARM, which are based on Von Neumann architecture, this device has a Harvard architecture, and simple bus interface eases connection to either cached or SRAM-based memory system. The features of ARM9TDMI processor relevant to the Speech Encoding and Decoding process are:

- Fifteen general-purpose registers out of which one register used while branching;
- Conditional execution of instructions;
- Five stage pipeline and Harvard architecture;
- Block data transfer Instructions;
- Instructions MUL (multiply) and MLA (multiply and accumulate) with source operand dependant execution cycle times;

3. SPEECH CODING METHODS AND IMPLEMENTATION ISSUES

Over the years speech compression algorithms have evolved to provide good quality speech even at sub 8 kbps rates. Prominent among them are Code Excitation Linear Prediction (CELP) based speech codecs. In which the residual signal obtained after linear prediction (LP) analysis on the input speech signal, is used to choose the optimal excitation code vector by employing analysis-by-synthesis configuration. With the advent of algebraic CELP, speech coding has become more computation intensive, while easing static memory requirements. These compression methods have been standardized in the last few years by ITU-T, ETSI and TIA in form of various proposals.

Speech compression algorithms like most of the other DSP algorithms are multiply-accumulate (MAC) and looping intensive. These algorithms are instrumental in the evolution of digital signal processors. Because of their specialized features like single cycle MAC capability, zero overhead looping, specialized addressing modes, hardware support for fixed point arithmetic and parallel address generation units have made them the most suitable platforms for implementation of wide variety of DSP based algorithms. With fast interrupt response, larger memory addressing capabilities and inherent architectural advantages for efficient compiler designs have made RISC processors more apt for control code flow management and operating systems.

The huge market for flexible, low power and low cost single chip solutions for high volume consumer applications has prompted many RISC processor vendors to provide limited [4] DSP functionality. For example ARM9TDMI provides an 8-bit by 8-bit booth's multiplier. Also the native data path width of most RISC processors today is 32 bits. While this extended precision over 16 bit DSP processors is an advantage for applications such as audio, it is a problem for standardized speech codecs. The speech coding standards target 16 bit DSP processor implementations and impose strict bit compliance. Software codes provided for bit exactness verification by the standard bodies is the simulation of 16 bit fixed-point arithmetic. Optimizations in typical RISC implementations are targeted at the efficient use of the multiplier along with other hardware features; looping overhead reductions, load reductions for slower memory access systems etc. These methods are discussed in detail in the next section.

4. RISC OPTIMIZATION TECHNIQUES

Along with the documentation describing the algorithms and references, the standard bodies provide an ANSI C code corresponding to a fixed-point fractional arithmetic implementation of the codec. A set of test vectors comprising input and output streams are given to verify the bit exact development of the codec on the target platform. The ANSI C code is written in a manner to aid porting to different platforms and in order to achieve this the basic mathematical operations of addition, subtraction, multiplication etc. are implemented as separate functions. This is done so that these operations can be rewritten in the most optimized manner for the specific DSP architecture. Also since the C language does not support fixed-point fractional arithmetic all the mathematical operations are simulated and functionalized. Since the RISC architecture is very different from that of a typical DSP it becomes necessary to modify the standard code and generate a working reference code. The modifications performed while developing the reference code addresses the issues like memory load optimization, reducing function call overhead, fixed-point arithmetic optimization, MAC optimizations, compiler-specific optimizations and loop optimizations.

4.1. Memory Load Optimization

One of the major techniques in developing the reference code is to develop the code at the processors most optimized memory load width. This width would generally be more than 16 bits, the precision at which the standard code is written. To maintain bit exactness with the standard, no extra precision is added and data width increased with the use of extra sign bits. The data is converted back to 16-bit standard code precision during the final write to the output stream. There is a considerable cycle gain achieved by generating the reference code with this method since it reduces the stall and wait cycles associated with sub word loads. As an example for the ARM9TDMI half-word (16-bit) loads induce pipeline stalls if the loaded registers are used immediately. Now since we convert the same 16-bit data into 32 bits without adding any extra precision and do word length loads we do not get any stalls. Further in ARM9TDMI multiple load and store instructions are available but operate only on word data and not on half-word data. So we get considerable code size advantages as well. From our implementation studies we found that the resulting increase in data memory size requirement is more than compensated by the corresponding reduction in program code size requirement. Program code size reduction has considerably more advantages that offset the increase in data memory size due to the data width increase. Since the program size is substantially larger than the program cache a reduction in the program size would reduce the program cache misses more than the increase in the data cache misses due to increased data memory requirements. In our experiments with a real-time device, while data cache hits remained almost the same, program cache hits increased substantially.

4.2. Arithmetic Operation Optimization

As explained earlier the standard code implements every fixed-point arithmetic operation by way of separate functions, to aid porting and profiling. These fixed-point arithmetic functions are hand assembled for efficient implementation on the target processor and inlined to reduce function call overhead. In fixed point arithmetic implementation saturation and overflow checks are an important set of operations. The standard code simulates these operations with additional checks and comparisons. While fixed point DSP's have extensive hardware support for these operations leading to almost zero overhead implementation, RISC processor implementations are cycle consuming operations. Thus it is a good idea to reduce these checks wherever they are redundant, by using the additional knowledge available about the data values and their ranges. For e.g. shift operations involving a constant shift value can avoid range compliance checks. Low overhead instructions for conditional instruction execution based on status flags can be effectively used on RISC processors for efficient implementation of these saturation and overflow checks.

Lack of single cycle multiplier/MAC units in RISC processors make multiplication and divisions costly operations relative to their DSP implementations. Division/Multiplication with dyadic numbers are converted into corresponding shift operations. In cases of constant non-dyadic multipliers, multistage combinations involving shift and ADD/SUB operations can replace multiplication. For e.g. multiplication by 40 can be split up into two stages, first stage involving independent shifts by 5 (multiply by 32) and 3 bits (multiply by 8) followed by a second stage addition operation.

The complier performance for RISC processors is substantially better than those for DSP processors. The performance for our working reference code can be still improved by following some simple steps. By the judicious use of local variables compiler performance can be improved considerably. We can try reducing the number of local variables used in specific functions. The code performance improves significantly since the complier is now able to dedicate registers to variables. Intensive loops from critical functions can be replaced by hand-assembled functions; the code performance improves although at the cost of additional functional overhead. Pointers in local variables used in critical loops, should be avoided as it necessitates the use of a memory location for the variable instead of a register. Limiting the number of variables passed in functions can reduce function-calling overheads. For functions with large number of arguments, suitably encapsulating the variables into a single data structure and passing its address reduces call overhead.

4.3. Loop Optimization

After the development of the reference code critical portions of the algorithm are hand assembled. Profile results of the C code provide a good estimate of the relative importance of functions. RISC assembly coding when compared to DSP assembly coding is relatively easy. But specialized techniques have to be applied to obtain the best performance of the hand-assembled code. Since DSP codes are looping intensive all DSP processors provide zero overhead looping (hardware loop), where the loop count update and check are performed in hardware without any cycle consumption.

Looping overhead in software loops is considerable and it is directly proportional to total iterations in the loop. Consider the example of the correlation calculation code in figure 2. Reduction in the looping overhead of the

```c
/*-------------------------------------------------------------------*
 * Procedure Autocorrelation:                                        *
 *                                                                   *
 * Compute the Autocorrelation of the Input data with varying lag    *
 *-------------------------------------------------------------------*/

void AutoCorr(
    Word32 lScaledSig[],/*Scaled Input Signal (Pre-Extended)*/
    Word32 lCorr[],     /*Autocorrelation Output Result     */
    Word32 Lag_Max,     /*Maximum Lag Value                 */
    Word32 lFrameLen    /*Input Data Length                 */
)
{
    Word32 i, j;
    Word32 lTemp;

    /* Compute all the correlations */
    for(i = 0; i <= Lag_Max; i ++)
    {
        lTemp = 0L;
        for(j = Lag_Max; j < lFrameLen + Lag_Max; j++)
        {
            lTemp += ((lScaledSig[j] * lScaledSig[j - i]) << 1);
        }
        lCorr[i] = lTemp;
    }
    return;
}
```

Figure 2. Correlation Calculation C code.

internal loop would give significant cycle count reduction. This can be achieved by dividing the internal loop count by a suitable factor (2-5) and performing that many instructions inside the loop, effectively reducing the number of jumps and loop end checks.

Another powerful technique for improving performance is reducing the number of loads in loops. This technique is well applicable to many DSP algorithms such as filtering and correlation calculations. Consider the example of a correlation calculation C code in figure 2 again. For every outer loop iteration we load lScaledSig[j] data again once. If the outer loop is unrolled then along with the reduction in looping overhead of the outer loop we can also use the first loaded value of lScaledSig[j] for the rest of the unrolled portions. When outer loop unrolling from this example is combined with inner loop unrolling we can achieve significant reductions in loads.

```c
    /* Compute all the correlations */
    for(i = 0; i <= Lag_Max; i +=4)
    {
        lTemp1 = 0L;
        lTemp2 = 0L;
        lTemp3 = 0L;
        lTemp4 = 0L;

        for(j = Lag_Max; j < (lFrameLen + Lag_Max); j +=5)
        {
            lTemp1 += ((lScaledSig[j + 0] * lScaledSig[j - i]) << 1);
            lTemp2 += ((lScaledSig[j + 1] * lScaledSig[j - i]) << 1);
            lTemp3 += ((lScaledSig[j + 2] * lScaledSig[j - i]) << 1);
            lTemp4 += ((lScaledSig[j + 3] * lScaledSig[j - i]) << 1);

            lTemp2 += ((lScaledSig[j + 0] * lScaledSig[j - i - 1]) << 1);
            lTemp3 += ((lScaledSig[j + 1] * lScaledSig[j - i - 1]) << 1);
            lTemp4 += ((lScaledSig[j + 2] * lScaledSig[j - i - 1]) << 1);

            lTemp3 += ((lScaledSig[j + 0] * lScaledSig[j - i - 2]) << 1);
            lTemp4 += ((lScaledSig[j + 1] * lScaledSig[j - i - 2]) << 1);

            lTemp4 += ((lScaledSig[j + 0] * lScaledSig[j - i - 3]) << 1);

            lTemp1 += ((lScaledSig[j + 4] * lScaledSig[j - i + 4]) << 1);

            lTemp1 += ((lScaledSig[j + 3] * lScaledSig[j - i + 3]) << 1);
            lTemp2 += ((lScaledSig[j + 4] * lScaledSig[j - i + 3]) << 1);

            lTemp1 += ((lScaledSig[j + 2] * lScaledSig[j - i + 2]) << 1);
            lTemp2 += ((lScaledSig[j + 3] * lScaledSig[j - i + 2]) << 1);
            lTemp3 += ((lScaledSig[j + 4] * lScaledSig[j - i + 2]) << 1);

            lTemp1 += ((lScaledSig[j + 1] * lScaledSig[j - i + 1]) << 1);
            lTemp2 += ((lScaledSig[j + 2] * lScaledSig[j - i + 1]) << 1);
            lTemp3 += ((lScaledSig[j + 3] * lScaledSig[j - i + 1]) << 1);
            lTemp4 += ((lScaledSig[j + 4] * lScaledSig[j - i + 1]) << 1);
        }
    lCorr[i] = lTemp1;
    lCorr[i + 1] = lTemp2;
    lCorr[i + 2] = lTemp3;
    lCorr[i + 3] = lTemp4;
    }
```

Figure 3. Correlation Calculation Modified C code.

In figure 3 we demonstrate the unrolling of the outer loop by 4 and the inner loop by 5 and subsequent data rearranging. We are able to further reduce loads since along with lScaledSig[j] values reuse, the values of the second multiplication operand are also the same for various values of i and j. As an example lScaledSig[j-i] is same as lScaledSig[(j+1)-(i+1)], lScaledSig[(j+2)-(i+2)] and lScaledSig[(j+3)-(i+3)]. Comparing the total number of loads from the two examples we have the loads in the original code (1+1)*lFrameLen*(Lag_Max+1) reduced to (5+8)*(Lag_Max/4+1) * (lFrameLen/5). Assuming values of 160 for lFrameLen and 100 for Lag_Max we see that we have reduced the loads from 32320 to approximately 11000. This load reduction assumes more significance since in many devices because of usage of slower memories to reduce cost, there are high latencies associated with memory.

5. RESULTS AND CONCLUSIONS

Reference code development and loop optimization techniques suggested in Section 4 were applied to the GSM-AMR [5] speech codec standard code. The development was carried out on the widely popular ARM9TDMI RISC processor. GSM-AMR with its variety of bit rates is the ideal candidate for voice content in Multimedia Messaging. Table 1 lists the cycle counts achieved for various modes of operation of the GSM-AMR codec in our development. The cycles quoted are assuming zero wait states in memory access. The program memory and constant table memory size about 100k bytes and the maximum stack usage was less than 8k bytes. The codec was verified for bit exactness with the standard reference streams. Table 1 also lists comparisons with Emuzed's Decoder implementations on the TI C55x and ADI Blackfin DSP processors.

Rate	Encoder	Decoder			Codec
		ARM	C55x	Blackfin	
4.75	42.40	6.81	1.36	1.36	49.21
5.15	33.80	6.85	1.35	1.35	40.65
5.90	38.20	6.88	1.36	1.36	45.08
6.70	44.66	7.03	1.36	1.37	51.69
7.40	42.10	6.18	1.22	1.20	48.28
7.95	44.88	6.67	1.29	1.28	51.55
10.20	43.06	6.33	1.28	1.30	49.39
12.20	44.10	6.38	1.34	1.28	50.48

Table 1. Cycle counts in Mega Cycles for various modes of GSM-AMR.

In this paper, we presented methodologies for real-time speech codec implementation on RISC processor architectures. The specific codec implemented was GSM-AMR on the ARM9TDMI processor. These methods are well applicable to any speech codec and RISC processor platform.

6. REFERENCES

[1] Nokia Networks, "Nokia Multimedia Messaging," http://nds1.nokia.com/press/background/pdf/feb01.pdf Nokia Mobile Phones, pp. 1-8, Feb 2001.

[2] O. Gunasekara, "Developing a digital cellular phone using a 32-bit Microcontroller," White Paper, http://www.arm.com/support/White_Papers?OpenDocument, Arm Ltd, pp. 1-7.

[3] ARM Ltd., *ARM9TDMI Technical Reference Manual,* Arm Ltd, ARM DDI 0145A, November 1998.

[4] Berkeley Design Technology Inc., *Inside the ARM7, ARM9 and ARM9E,* http://www.bdti.com/products/reports_arm.htm.

[5] Universal Mobile Telecommunication System (UMTS); Mandatory Speech Codec speech processing functions, "AMR Speech Codec – Transcoding Functions," 3GPP TS 26.090, Version 4.0.0, Release 4, March 2001.