White Paper
SPRAAJ7–July 2008

Programming of Audio Coprocessor for MP3 Decoder

Ramesh Naidu G., Fitzgerald J. Archibald, Stephen H. Li

Audio playback requires long battery life in handheld applications like portable audio players. To achieve low power and die area, the number of gates needs to be reduced. An audio coprocessor can help in achieving low power and die area. The audio coprocessor/accelerator needs to be programmable to address multiple codec standards (new and existing), and numerous pre- and post-processing algorithms. This paper describes assembly programming techniques of an audio coprocessor for the MP3 decoder for better utilization of hardware and the optimization of cycles and memory.

Index Terms: Assembly programming, audio processor, low power audio engine, MP3 decoder.

Contents

List of Figures

1 Introduction

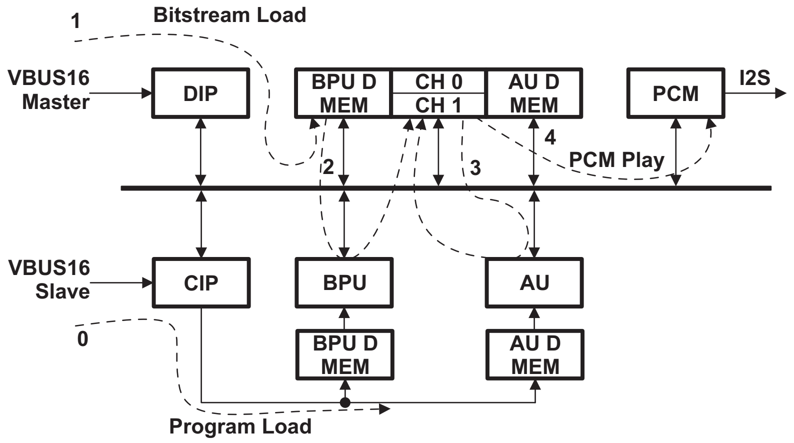
Embedded programs are developed using high level languages like C to leverage code maintainability, portability, and readability. Program development time is lower for C programs compared to assembly programming because of the availability of development tools like a cross-compiler, linker, and source level debugger.

Embedded programs often use assembly programming for optimizing cycles and memory. In addition, assembly programming is useful to have better control over hardware in the case of device drivers. This paper provides a study of the use of assembly programming for developing audio codecs in an audio processor. In addition, the firmware development cycle of the MP3 decoder development is presented.

2 Hardware Overview

The functional block diagram of the audio core is shown in Figure 1. The audio core is composed of two autonomous processing units: the Bit Processing Unit (BPU), and the Arithmetic Unit (AU) and I/O peripherals. The BPU and AU interface through shared memory. The BPU is essentially a control processor with special instructions and hardware for bitstream parsing [1]. The AU is a programmable fixed-point computation engine for performing DSP operations like Discrete Cosine Transform (DCT) and filters. Core and I/O peripheral synchronization are carried out by the BPU (master). The audio core can be used as a coprocessor for audio codecs and effects like an equalizer. The programmable hardware resources are memory (RAM and ROM), Direct Memory Access (DMA), IO ports, and interrupts.

Figure 1. Audio Processor Block Diagram



The audio core is a slave requiring a master (host) for initialization. The master could be a general purpose processor like ARM. While holding the audio core in reset (halting processor execution), host loads programs of BPU and/or AU (flow 0). BPU can load programs onto AU. For reducing program memory size, memory overlay can be used [4]. That is, only the program needed for audio processing, for a given input stream, is loaded onto the audio core memory. The program/data memory is loaded dynamically based on need. Though dynamically loaded, the dynamic loading is per stream decode/encode. Specifically, program/data is not dynamically loaded for each frame in the stream.

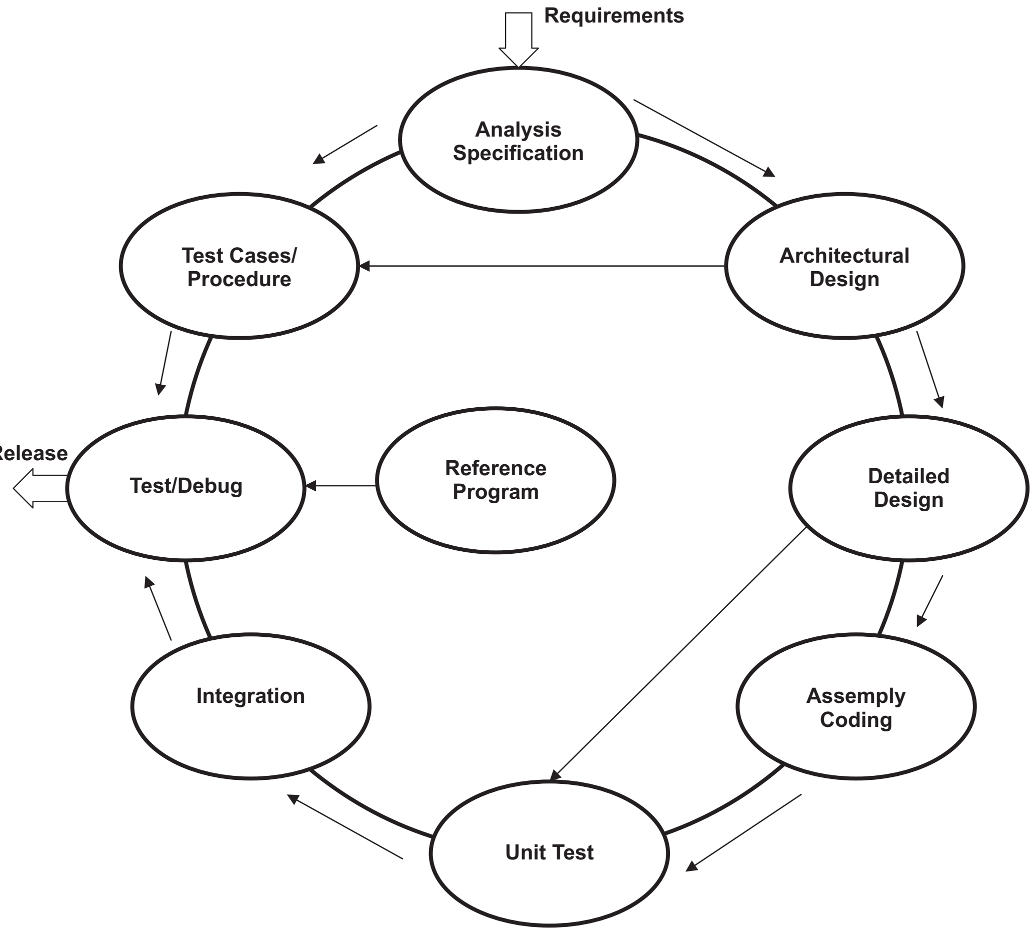
Consider Figure 1 with respect to MP3 decoder. Host shall fetch the streams for decode onto external memory, which can be Synchronous Dynamic RAM (SDRAM) from hard-disk or flash memory. Host would inform the data start, and size parameters to audio core via Control Input Port (CIP). BPU can program Data Input Port (DIP) DMA to fetch the streams onto internal memory of audio core (flow 1). BPU unpacks the bitstream and decodes variable length (Huffman decode) encoded information. BPU writes the control information extracted from bitstream and Huffman decoded frequency samples into shared memory (flow 2). BPU gives control to AU for generation of Pulse Code Modulation (PCM) samples from frequency samples. AU performs de-quantization, joint-stereo, anti-alias butterfly, Inverse Modified Discrete Cosine Transform (IMDCT), overlap-add, frequency inversion and synthesis [3]. The synthesized PCM samples are written into shared memory (flow 3). Once the PCM samples are available in shared memory, AU gives control to BPU for streaming the samples to Digital to Analog Converter (DAC) via PCM port (flow 4).

3 Firmware Development

3.1 Development Life Cycle

The waterfall development life cycle, shown in Figure 2, is followed in MP3 decoder development [6].

Figure 2. Development Life Cycle



3.1.1 Requirement Analysis

Functional requirements are specified by the ISO/IEC 11172-3 standard [6]. Performance requirements like cycles, memory, streaming, and host interfaces are specified based on target application. Implicit requirements cover aspects like maintainability, and code readability. The requirements stated explicitly and implicitly are fed as input for test case development.

3.1.2 Architectural Design

The top level design involves designing interfaces between BPU, AU and IO peripherals. Also, top-level design needs to take care of real time streaming and buffering. The interfaces for interaction with host are defined here as well. Architectural design could be viewed as system or subsystem design with details to timing, streaming (buffering) and interfacing. Major modules are defined during architectural design. Architectural design provides inputs for preparing test cases and procedures for testing interfaces.

3.1.3 Detailed Design

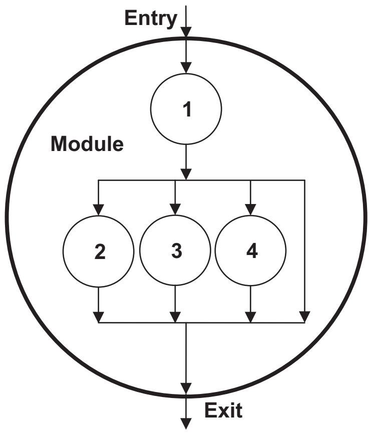
This step involves the design of modules and sub-modules like parser, and Huffman decoder. Module interfaces are defined for integration of all modules into a codec. The design representations could be pseudo-code, flow chart, state diagrams, etc. for individual modules. The design inputs are used for defining unit test cases and procedures.

3.1.4 Assembly Coding, Unit Test

The detailed design of modules and system are translated to assembly language programs. The translated code sections are unit tested for functionality, interface, memory usage, cycles profiling, parameter range, and memory overwrites. Code is reviewed for correctness of implementation, comments, efficiency/optimization, and conformance to coding guidelines.

The best way to develop assembly program is to modularize. Modular programs are easy to implement, test, debug, and maintain. System is subdivided into subsystems, which are refined further into modules based on functionality and hardware partitions as shown in Figure 3. Module A is sub-divided into modules 1, 2, 3 and 4. The module size has to be optimal while partitioning code based on functionality, to avoid call overhead. Every module shall have singly entry and single exit following structured programming.

Figure 3. Structured Program



Well commented code along with design documentation is easy to maintain. Coding style and indentation is critical for readability of assembly programs.

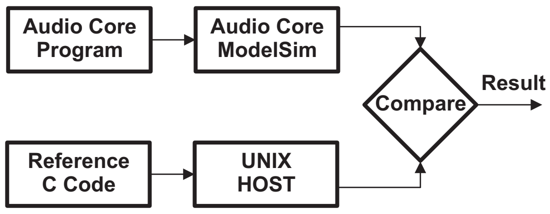
3.1.5 Integration

The BPU modules are integrated to form BPU subsystem. The subsystem is tested for correctness of bitstream decode and Huffman decode. Lossless decoding modules are tested for bit exactness using reference decoder. The interface conformance of BPU subsystem is tested. BPU cycles and memory usage are benchmarked. Similarly, AU modules are integrated and tested. The BPU and AU systems are integrated. The integrated decoder is tested. Finally, the audio core program is integrated with host program and tested.

3.1.6 Test

The system is tested for requirements, stability, error handling, long duration operation, interface and performance bench marking. The test platform consists of ModelSim simulator executing audio decoder on audio core and a reference decoder running on workstation, shown in Figure 4. The output of reference decoder and audio core are compared for Root Mean Square (RMS) error and ISO/IEC 11172-3 compliance [8].

Figure 4. Test System Setup



3.2 Memory Management

RAM occupies relatively larger area of the die compared to ROM. So ROM is preferable over RAM [4]. Overall, the memory (ROM and RAM) has to be minimal to reduce die area. Assembly programming provides better control over memory and instructions. Thus program and data memory footprint is optimized by hand assembly. Trade-off can be made for speed versus memory size for some cases.

ROM is used for constant data and stable programs. Data is multiplexed within a word to achieve memory optimization. ROM is shared across modules when feasible.

Data RAM is optimized by packing multiple variables into single memory location. For example, an array of four elements with element size of 4 bits data can be packed into a single 16-bit memory location. The data element is unpacked, when needed. This pack/unpack operations consumes extra cycles during program execution.

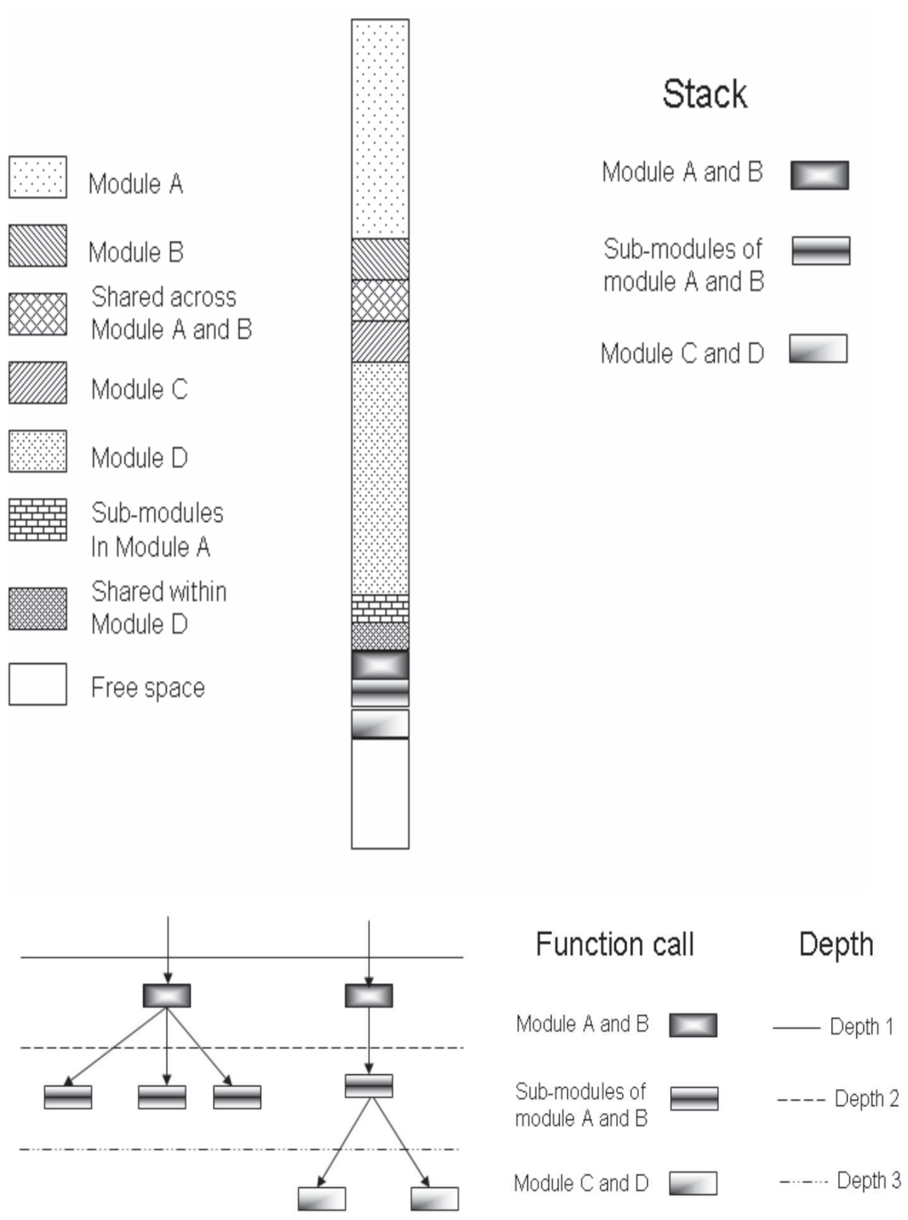
Data RAM is optimized by sharing memory locations across modules. At any point in time, only one module will be accessing the shared memory. This typically requires that the modules be at same call depth level and have mutually exclusive operation. By this method, statically allocated memory can be made to behave like dynamically allocated memory without the consequences of memory fragmentation and leak. Resource sharing mechanisms like semaphores from Real Time Operating System (RTOS) are not needed, since programs on audio core are single-threaded [7]. Care should be taken to avoid sharing memory across modules and interrupt service routine (ISR).

Program memory is optimized by the use of modules. MACROS for code repetition are used instead of function calls to optimize call overhead. MACROS need to be small size modules.

Stack: Audio core does not have hardware defined stack. Simplest way to create stack without degrading performance for single-threaded applications is to have stack memory defined per module. On entering a function, the registers are saved into the module stack memory. On exiting the module, the register contents are restored with context saved in stack memory. Stack memory can be used for local variables inside the module. Stack memory is shared across modules just like the heap memory previously discussed. The program shall maintain independent stack for caller and callee modules, irrespective of direct caller/callee relationship.

Figure 5 illustrates data memory sharing and stack memory sharing across various modules. The depth of function call tree is three and three stack memory regions are used for managing stack operations. The functions A and B are at same depth (depth1) and so the stack memory of A can be shared with B. The maximum stack size required by A or B should be used for allocating stack memory. Similarly, the sub modules of A and B share the same stack memory as they are at same depth (depth2). Specifically, the memory is released by module A to module B for usage, when module A has finished accessing memory. The modules C and D share the same stack area as they are at same depth (depth3).

Figure 5. Memory Sharing and Stack Layout



The code snippet below shows the simulation of stack in module A.

```
! Code snippet for saving r1, r2, r3 registers of function A
!
   st(r1, mem(Stack1_r1_save))       !r1 save
   st(r2, mem(Stack1_r2_save))       !r2 save
   st(r3, mem(Stack1_r3_save))       !r3 save
   ...
   ...
   ! Restore r1, r2, r3
   ld    (mem(Stack1_r1_save), r0)
   ldpar (mem(Stack1_r2_save), r0)   ld (r0, r1)   !restore r1
   ldpar (mem(Stack1_r3_save), r0)   ld (r0, r2)   !restore r2
   ld    (r0, r3)                                  !restore r3
   ...
   ...
```

3.3 Interrupt Handling

Interrupts are used to reduce the overhead of software polling. On completion of an activity by hardware resources like DMA transfer completion, interrupt request (IRQ) is registered. ISR associated with the IRQ is invoked by the BPU. Keep the ISR execution time minimal by clearing interrupt and storing the event in order to meet real-time constraints. The event can be checked in program. Avoid using loops and polling in the ISR. In case of PCM output (I2S DMA) interrupt, the output buffer for streaming is loaded onto DMA in the ISR [2].

Consider the case of byte aligned sync search operation in MP3 decoder. When sync word is not found till the buffer end, an interrupt is triggered. In ISR, the buffer has to have new data to resume the sync search operation, as entire data in the buffer is parsed or consumed by sync search operation. Since the hardware does not allow nested interrupts and unavailability of RTOS, triggering the DMA source for refill and checking for DMA completion is not possible. To handle this case, unconventional branch (extension module) is used, instead of normal return from ISR. Before taking the branch to ISR extension module, context of sync search routine and ISR return address are saved in reserved memory (not shared with other modules) and DMA transfer is initiated to refill the buffer. After the refill is over, ISR is invoked and ISR sets an event for the extension module and returns normally. The extension module comes out of the event polling and restores the sync search context from reserved memory and returns back to the interrupt return address of sync search operation.

3.4 Cycles Optimization

Cycles can be optimized by usage of instruction parallelism, apart from algorithm optimization and coding efficiency. Pipelined architecture reduces cycle time of a processor and hence increases instruction throughput. For instance, n-stage pipeline would take (n+k-1) cycles for executing k instructions. Instructions in delay slots of branch instructions are executed irrespective of whether branch is taken or not. The number of delay slots depends on the pipeline stages. For n-stage pipe line, (n-1) delay slots are available. Programs have to make good use of delay slots for optimization of cycles.

The code snippet below is for a two stage pipeline, as in BPU, which does a memory read and ALU operation in one clock cycle.

```
ldpar (mem (my_memory_location), r0)    add   (r1, r2)
```

Table look up and pre-computed tables are used to optimize cycles consumption. Fixed-point arithmetic is used to perform DSP intensive operations on ALU [5].

Code partitioning between BPU and AU is essential to balance load while keeping BPU and AU interaction to minimum. BPU and AU cores can go into IDLE mode independently to save power.

4 Summary

The illustrated programming techniques aid in realizing stack memory and interrupt handling to develop modular programs with limited support from processor. The cycle and memory optimization is achieved by use of assembly language programming of the audio coprocessor. Assembly language programming allowed for better utilization of hardware resources and instruction level parallelism provided by the audio coprocessor.

5 Acknowledgment

The authors thank Mike Polley, Architecture Team, TI for sharing his thoughts on the core architecture. Authors value the contributions of Mohamed Mansour, R&D, TI towards algorithm and software.

6 References

1. *An AC-3/MPEG multi-standard audio decoder IC* by S. H. Li, J. Rowlands, P. Ng, M. Gill, D.S. Youm, D. Kam, S.W. Song, P. Look, CICC 1997, Proceedings of the IEEE 1997 Volume , 5-8 May 1997, pp. 245 - 248.
2. *Audio system for portable market* by F. J. Archibald, AES Preprint 6906, Convention 121, Oct 2006
3. ISO/IEC 11172-3:1993 Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 3: Audio.

4. *System-on-a-chip for portable audio player* by F. J. Archibald, M. O. Polley, S. H. Li, J. D. Kridner, ICCE 2007, pp. 1-2, Jan 2007.
5. *Fixed-point arithmetic: An introduction* by R. Yates. Available: http://home.earthlink.net/~yatescr/fp.pdf
6. *Software engineering—A practitioner's approach* by R. Pressman, 3rd ed., New York: McGraw-Hill, 1992, ch. 3.
7. *Real time systems: A tutorial* by F. Panzieri, R. Davoli, Technical Report UBLCS-93-22, 1993.
8. ISO/IEC 11172-4:1995 Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 4: Compliance testing.

Ramesh Naidu was born in Anantapur and received BTech in electronics and communications engineering, from Intell Engineering College, Anantapur, India in 2004. This author completed MTech in telematics and signal processing from NIT Rourkela, India from July 2005-June 2007.

He has completed MTech project work in Texas Instruments, Bangalore, India in the role of project trainee.

Fitzgerald Archibald earned a B.E (Electronics and Communication Engineering) from PSG College of Technology, Coimbatore, Tamil Nadu, India in 1996. He worked on control systems software development for geo-synchronous satellites from 1996 to 1999 in ISRO Satellite Centre, Bangalore, India. In 2001-2002, he worked on speech decoder, real-time kernel, and audio algorithms for DVD audio team in Sony Electronics, San Jose, USA. While in Philips Semiconductors (Bangalore, India, and Sunnyvale, USA) in 1999-2001 and 2002-2004, he worked on audio algorithms and systems for STB, DTV, and internet audio. He is part of the Personal Audio Video and Digital Imaging groups in Texas Instruments Inc, Bangalore, India from 2004-till date working on audio, video, and imaging systems and algorithm development. Interests include multimedia and control algorithms and systems.

Mr. Archibald is member of AES.

Stephen Li earned his BS in Electrical Engineering at University of Texas at Arlington, MS in Biomedical Engineering at UT Arlington/UT Southwestern Medical Center, and MS in Applied Mathematics at Southern Methodist University.

He joined Texas Instruments in 1984, has worked in different capacities on the design, modeling, processing, application, and architecture definition of VLSI IC. He is the author of several technical papers and journal articles covering VLSI architecture and design. He owns over 25 U.S. patents in the same area, as well as A/V algorithm development.