Audio Video Algorithm Porting on Different Tool Chains

Ketan Patel

Audio-Video algorithm porting on different tool chains

ABSTRACT:

Today audio-video algorithms are highly used as a communication medium. It is the foundation of the entertainment world. All these audio-video algorithms are initially developed on a particular platform. ARM is one of the cores that are used by mobile devices as well as by different tool chain environments such as Green Hills MULTI, ARM Developer Suite and Linux with arm patch. All these tool chains compile and link with their own programming structure. However, these are exceptions. It is a difficult task to develop and handle different versions of code for different versions of tool chains. In the embedded world, most of the code is built with assembly language thus making it difficult to re-write assembly code for different tool chains. A panacea for this would be a patch that would make the library work on different tool chains.

ARM libraries/binaries (elf) have a particular structure. This structure is quite complicated. Building patch involves understanding of library/binary structure and understanding of compiler and linker functionality for different tool chains. This patch will reduce coding efforts and maintenance for audio-video algorithms.
This paper covers information about ARM libraries/binaries and functionality of compiler and linker. It focuses on the instructions that cause libraries/binaries to fail to work on different tool chains and also on how these problems are resolved.

KEYWORDS:

Audio-Video algorithms, ARM processor, Compiler, Assembler, Linker, Symbol table, Relocation table.

INTRODUCTION:

Porting of audio-video algorithm can be described as a means of translating software targeted at a source execution environment into a form that can be executed in a destination environment. Once the audio-video algorithm is built on a particular platform, modifying its assembly and C code to support other tool chains is quite a hectic job. Further copies for all tool chains need to be modified whenever base algorithm is modified.

Different tool chain environments use different mnemonics. This would result in a lot of effort being put to translate the source code's assembler directives and mnemonics for different tool chains. The difference between different versions of tool chains lies among compiler, assembler and linker. The linkers of different tool chains will not understand compiled and assembled object code while linking. The definition or meaning of symbols generated by compiler and assembler of different tool chain are different. In order to port library they should be modified to a common definition for all linkers. For this modification, behavior of compiler, assembler and linker of different tool chains must be understood carefully.

ARM ARCHITECTURE:

ARM instructions are uniform and have 32 bit fixed-length instruction fields to simplify instruction decode. Out of 32 bits, the last 24 bits are offset for branch instructions. ARM incorporates the following typical RISC architecture features.

- Large uniform register file
- Load/store architecture
- Simple addressing mode
- Uniform and fixed-length instruction fields

It also provides the following additional features:

- Control over both ALU and shifter in every data-processing instruction to maximize the use of an ALU and a shifter.
- Auto-increment and auto-decrement addressing modes to optimize program loops.
- Loading and storing multiple instructions to maximize data throughput.
- Conditional execution of all instructions to maximize execution throughput.

ARM Registers:

ARM has thirty one general-purpose 32-bit registers. At any one time 16 of these registers are visible. The other registers are used to speed up exception processing. Out of 16 visible registers R13 is stack pointer, R14 is link register and R15 is program counter. ARM also has Current Program Status Register that gives information about condition code flags, interrupt disable bits, bits that encodes current processor mode and bits that encode ARM or THUMB mode. Thumb mode is 16 bit instruction.

ARM Instructions:

ARM instructions are divided into six classes. They are Branch instructions, Data processing instructions, Status register transfer instructions, Load and store instructions, Coprocessor instructions and Exception generation instructions. Of these six classes of instruction, porting effort needs to concentrate on branch instructions. Other instructions are almost the same in execution for all ARM platforms.

ARM Data Types:

ARM supports 8 bits (Byte), 16 bits (Halfword) and 32 bits (Word) data types. All the data operations are performed on word quantities. Load and store operations are available for bytes, half words and words. ARM instructions are exactly one word and Thumb instructions are exactly of half word.

ARM Branch instructions:

All ARM processors support a branch instruction that allows a conditional branch forwards or backwards up to 32 MB. The branch and link (BL) instruction preserves the address of the instruction after the branch (return address) in the LR (R14) register. The branch and exchange (BX) instruction allows inter-working branches between ARM and THUMB mode.

Understanding of branch instruction:

ARM core handles branch instruction in the following way:

- Sign extending the 24 bit signed (2's complement) immediate to 32 bit.
- Shifting the result left 2 bits
- Adding result to the address of branch instruction (PC) plus 8.

So, if library built on one tool chain is linked to another tool chain, the branch instruction will jump to the address of function + 0x8.

Opcode of B and BL instructions:

Branch (B) or branch and link (BL) instruction creates a branch to a target address, and provides both conditional and unconditional changes of program flow.
The below figure represents the structure of the instruction:

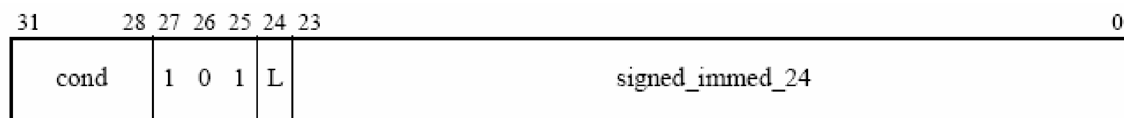

Figure 1: B,BL instruction

"L" is the 24th bit and it stores return address in a link register (LR).
"cond" is a condition under which instruction is executed. Condition fields can be equal, not

equal, greater than, less than, negative, overflow, carry etc.
Target address specifies the address to branch to.

Opcode of BX instruction:
The BX (Branch and Exchange) instruction branches to an address held in a register Rm, with an optional switch to thumb execution. The branch target address is the value of Rm register with its bit 0 forced to zero. The instruction set to be used at the branch target is chosen by setting the CPSR T (thumb) bit to bit 0 of Rm.
Following figure represents the structure of the instruction.

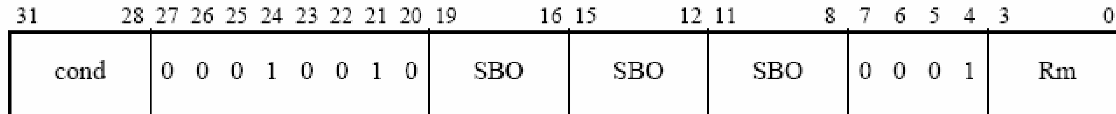

Figure 2: BX instruction

Rm is the value of branch target address. Bit 0 of Rm is set to 0 to select a target ARM instruction or 1 to select a target Thumb instruction.

ARM library symbol table:

For every function and variable, defined in a file, the compiler makes an entry in te symbol table. Symbol table holds information needed to locate and relocate a program's symbolic definitions and references. The symbols in library files convey specific information to the linker and the loader. Format of symbol table entry for ARM elf library is as under.

<base address> <Global/Local> <Function/Object/Data> <section> <size> <function name>

When compiler encounters any function call which is defined in a different file, it makes entry in the symbol table, and puts *UND* (undefined) in section field and 0x00000000 in base address field as shown in the following example.

ARM library relocation table:

Relocation is the process of connecting symbolic references with symbolic definitions. When the symbol is not found at compile time, compiler adds entry for that symbol in the relocation table. Relocation table entry for ARM elf library is as under.

<OFFSET>	<TYPE>	<VALUE>
4 bytes	1 byte	3 byte symbol + 4 byte offset

TYPE field in Relocation table:

R_ARM_PC13 (R_ARM_POOL) - 0x04
R_ARM_ABS32 - 0x02

R_ARM_PC24 - 0x01

A relocation section references two other sections: a symbol table and a section to modify. The section header's sh_info and sh_link members specify these relationships.

DESCRIPTION OF PORTING PROBLEMS:

The library generated on one environment has an offset in its branch instruction, which is not resolved by another tool chain environment. In any assembly file, if a function calls the other function of the same file, the offset generated by one tool chain cannot be resolved by the other tool chain environment. A patch has to be added in the output library in order to resolve this problem.

Mentioned below are some problem definitions and solutions for branch instructions to branch function calls and load/store instruction to load/store global variables and arrays. Also problem description and solution for symbol table setting for above issues are described below.

Problems with Branch instruction BL (conditional and unconditional):

Consider this:

One of the tool chains replaces some function calls by instruction, eb000000. It also makes entry in relocation table about this instruction and the type field in relocation table is set to R_ARM_PC24 (0x04). Linker finds entry in relocation table and adds actual address of function (from symbol table) to an offset in opcode.

Below is a comparison of two different compilers. Also illustrated is the problems created when we use compiler of one tool chain and linker of another tool chain.

	Compiler	Linker
Tool chain 1	Replaces branch instruction by eb000000 (24 bit offset = 0)	Resolves function address by, offset in opcode + Actual address of function – 8
Tool chain 2	Replaces branch instruction by ebfffffe (24 bit offset = - 8)	offset in opcode + Actual address of function

Table 1: Behavior of different tool chains

To resolve the above mentioned conflicts, the patch has to deduct or add difference of offset between different tool chains. For example, in the above case the patch will update the opcode eb000000 with ebfffffe so that it will easily link on tool chain 2.

Problems with Branch instruction b (conditional and unconditional):

Branch (B) to labels that have an entry in relocation table, may jump to any incorrect memory location when it is linked on different tool chains. On other hand, all branches (B) to labels that

do not have entry in relocation table jump to correct memory locations.
The incorrect address is calculated as follows:

Wrong branch address = (<offset in opcode> * 4) + 8 + <address of label (from symbol table)>

To resolve this problem remove the offset in opcode.
Doing so will enable the linker to get the address of the label from the symbol table and hence no offset will be added.

Problems with temporary labels (.L<number>):

The compiler does not replace all function calls by eb000000. These function calls will be replaced by ldr and branch instructions. Complier creates one temporary label name i.e. .L<number>, and makes an entry of this label in the relocation table. The type field of this entry is R_ARM_ABS32 (0x02).
ldr instruction loads value from a temporary label and branches to this value, assuming that loaded value is an address of a function. The linker finds an entry in the relocation table and replaces actual address of function at temporary label.

Relocation table contains entry for opcode where the label is defined. On the address where the label is defined some garbage value might be stored. While linking, this value is replaced with the sum of base address and the garbage value (offset).

To resolve this problem the opcode where the label is defined is replaced by the offset specified in relocation table entry. Linker will then add the offset (same as that available in relocation table) in base address of label.

Problems with symbol offset:

There are offsets available to jump to particular indices in places where symbols are defined. On some tool chain, value at label will be replaced by base address of array or function pointer table even though offset field contains valid offset value. So during linking time, linker will select the base address instead of the address at the specified offset from the base address.

This problem is resolved with the same approach as we described in previous case. Here if the label is defined in relocation table then the offset specified in relocation table is copied to the opcode. So linker will get the offset from the opcode and will load the value according the address specified in the relocation table.

CONCLUSION:

By changing the opcodes, entries in symbol table and relocation table it is possible to translate a library to support different tool chains. A simple patch to library will make it to support and link on different tool chains. This concept has been implemented and tested thoroughly. Further it has also found that this concept does not compromise with performance. Also it has been tested with different versions of Linux tool chain and it works correctly. It helps to deliver various audio-video codec libraries on different environments with very less effort which otherwise may

take almost double the effort to start from scratch.

REFERENCES:

ARM Developer Suite (ADS) document suite available on www.arm.com
http://www.x86.org/ftp/manuals/tools/elf.pdf - Executable and Linking Format (ELF) Specification (version 1.2)