

CS8803 Summer 2009
Language and Compilers for Embedded Systems
Project #1 Code Compression

Assigned on 06/03/2009 **Your code frozen before 06/24/2009**

Score: _____ Date: _____ Name: _____

TA Signoff: _____

The purpose of this assignment is to help you understand the implementation of code compression. There are two parts in this project. First, you will write a program to perform the CodePack-style code compression method using ARM-ELF binary files. The second part is to implement the Dictionary-Based code compression scheme using the VEX assembly file as the input. We will provide sample binary and assembly files for the purpose of your own debugging and test. During the check-off, the TA will use different test files for grading. For both parts of the project, your code should run on the Linux machine. (However, you can develop your codes on any platform so long as you can compile your solution on a Linux-based system for check-off.)

Part I.

1. (50%) CodePack-Style Code Compression. The first part of the project is to write a program that implements a code compression method similar to the IBM CodePack discussed in class. In short, the compression will take place for the higher 16-bit and the lower 16-bit separately for each instruction. Your program will compress the instructions based on their respective frequency of occurrences. Please read further for more detailed instructions. Also read the paper titled “CodePack: Code Compression for PowerPC Processors” or our lecture slide.

1.1. Download the sample.bin from the CS8803 course website. It is a sample ELF binary provided for you to debug and test. Note that executable instructions are contained in the text section of the ELF file. Thus, the first step of code compression is to extract data from the text section. Please print out the information of the text section in the ELF binary file using the following format.

Text Section Information
Start address: 0x0002f0
Size of section (bytes) : 0x004a0

- 1.2. Write a program to perform the CodePack-style code compression using the instructions in the text section. You need to do the following steps enumerated below. Please note that it is not required to generate the index table in this project. You are allowed to use any high level language and IDE you feel comfortable with, so long as they can be compiled under a standard Linux machine when you check off your code with the TA.
1. Count the frequency of the unique high 16-bit and low 16-bit instruction patterns that exist in the text section.
 2. Generate the lists for each high and low 16 bit patterns and sort them according to their frequencies from high to low.
 3. Build the decoding tables based on the sorted lists and the CodePack encoding format. (Please see the Hint section below.)
 4. Generate the complete codeword file replacing the high and low 16-bit patterns in the text section with the appropriate entry from decode tables.
 5. Print out the statistics of the CodePack code compression.
- 1.3. For your output, you should generate the compressed.out file containing compressed instructions and print out the following critical information after performing code compression.

CodePack Compression Statistics

Total number of instructions:

Total number of instructions with their high 16-bit compressed:

Total number of instructions with their low 16-bit compressed:

Total number of compression groups:

Number of unique high 16bit pattern:

Number of unique low 16bit pattern:

Total codeword size (bits):

Total decode table size (bytes):

Compression ratio (codewords only):

Hints: The following example illustrates the processes step-by-step to implement your program.

STEP 1: Generate two separate sorted lists from the instructions in the text section.

High 16-Bit Patterns	Count	Low 16-Bit Patterns	Count
1110111110101110	8	0000000000000000	12
1010110010110111	6	0001010100101011	5
1111001110110101	4	1110010101110111	3
1011010111010101	2		

STEP 2: Build two separate decoding tables.

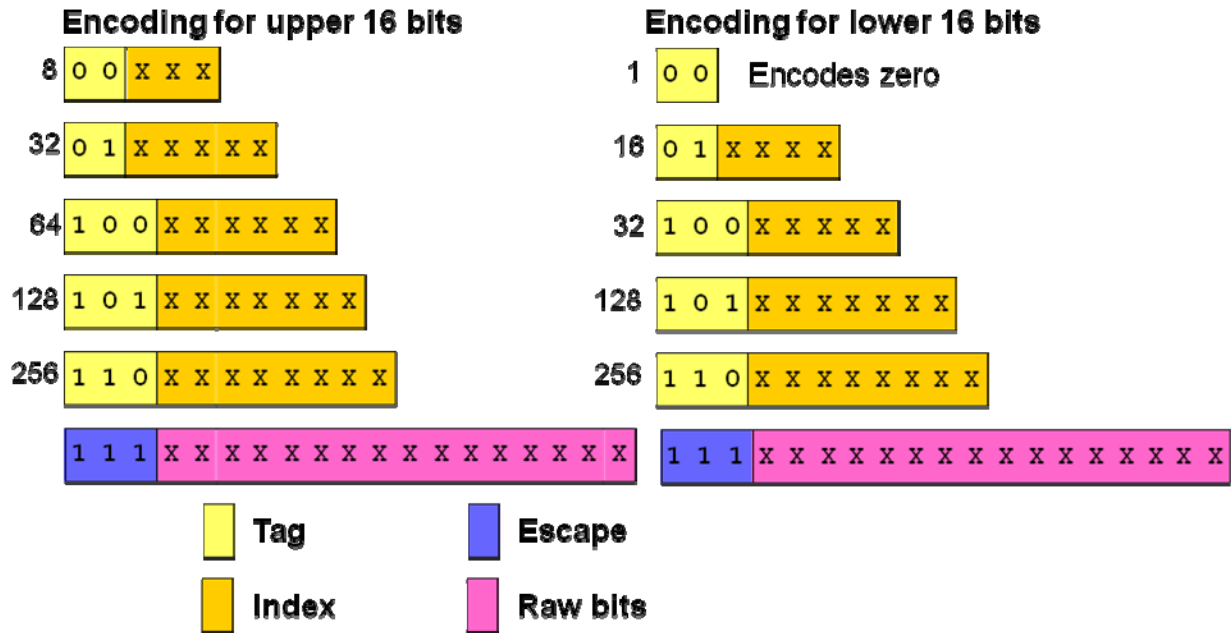
High 16-bit decoding table			Low 16-bit decoding table		
TAG	INDEX	DATA	TAG	INDEX	DATA
00	000	1110111110101110	00		0000000000000000
00	001	1010110010110111	01	0000	0001010100101011
00	010	1011010111010101	01	0001	1110010101110111
00	011	1011010111010101			

STEP 3: Generate the complete codeword for each instruction.

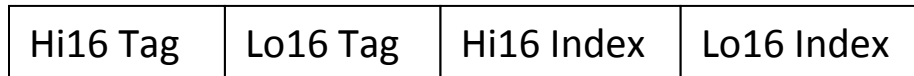
32 Bit Instructions	Codeword
1110111110101110 0000000000000000	00 00 000
1011010111010101 0001010100101011	00 01 010 0000
...

STEP 4: Create the file containing the complete codeword and print out the statistics following the format described in 1.3.

CodePack Encoding Format:



Codeword Format:



Part II.

2. (50%) **Dictionary-Based Code Compression.** For the second part of the project, you will implement a program performing the Dictionary-Based Code Compression method based on the paper titled “Improving Code Density Using Compression Techniques.”

2.1. Download the part2.tar.gz file on the CS8803 course website. The archive file includes the Linux static library used for generating a CFG from VEX assembly files, the sample.s VEX assembly file, and the sample source files which show how to use the library. You must run the program and make use of the library for the second part of the project.

2.2. Implement your program to perform the Dictionary-Based code compression method. You do not need to know the VEX assembly instructions for this project. Since the binary representation of the VEX ISA is unavailable, your program will perform the code compression based on the VEX assembly file. To prevent terminology confusion between VEX and the paper, we will follow the terminologies used in IA-64/Itanium Processor Family. In IA-64, an instruction is equivalent to an operation in VEX and an instruction bundle is an instruction in VEX. Basically, you need to do the following steps to complete Part II.

1. Read any give *.s file generated by the VEX compiler and collect basic block information.
2. Build the dictionary and make the codeword using the following rules.
3. Print out the critical information of the compression.

- Rule 1: Each VEX instruction is 4 bytes. (Not instruction bundle)
- Rule 2: Use fixed-length codewords of 2 bytes. The first byte is an escape byte and the second byte selects one of 256 dictionary entries.
- Rule 3: Each dictionary entry is limited to 16 bytes (i.e., 4 VEX instructions.)
- Rule 4: The entire 8 bits of the first byte can be used for escape bytes unlike the explanation of the section 4.1 in the paper. That is, we will have 256 escape characters, which can specify up to 65536 different codewords. (256 escape characters * 256 entries per an escape sequence)

For your output, you should to generate the dictionary.out file containing dictionary data and print out the critical information. Please note that you do not need to generate the compressed code and see the following formats.

dictionary.out

```
CODEWORD: 0x0000
ENTRY:
lbz      r9, 0(r28)
clrlwi  r11, r9, r24
addi    r0, r11, 1
cmplwi  cr1, r0, 8

CODEWORD: 0x0001
ENTRY:
lwz     r9, 4(r28)
stb    r18, 0(r28)
...
```

Compression Statistics

Total number of instructions in the original assembly file:

Total number of used codewords:

Dictionary size (bytes):

Compression ratio:

How to receive credit? TA Check-off is required for this project. (More details will be announced later.) You need to prepare a summary report for what you did in this project and show it to the TA for receiving credits.

Honor Code. This assignment must be done **individually**. For those who violate the rule, both the originator and the plagiarizer(s) will receive zero credit and will be **immediately** reported to the Dean of Students' Affair for further action.