Lesson 110: Huffman Encoding and Decoding

University of Victoria
British Columbia • Canada

Huffman (variable-length) coding

- Optimal encoding with respect to transmission rate
- Based on the probability of each symbol
  - Uses a variable-length code table for encoding a source symbol
  - The code-length depends on the probability of occurrence
- Let us assume a 5-symbol alphabet having the following probability distribution:
  A / 0.4, **B** / 0.3, **C** / 0.15, **D** / 0.1, **E** / 0.05
- Encode in a way that minimizes the transmission rate:
  - **A** – 0
  - All the others – 1
    - **B** – 0, that is **B** is 10
    - All the others – 1
      - **C** – 0, that is **C** is 110
      - All the others – 1
      - ...

Hufmann encoding

- The coding table:

Symbol	Bit combination	Code-length
A	0	1
B	10	2
C	110	3
D	1110	4
E	1111	4

- 3 bits are needed to represent the alphabet symbols
  - Transmission rate: 3 bits/cycle
- Between 1 and 4 bits are needed to represent the code-words
  - Transmission rate: 2 bits/cycle
    $(0.4 \times 1 + 0.3 \times 2 + 0.15 \times 3 + 0.1 \times 4 + 0.05 \times 4 \approx 2)$
- Penalty: sequential (slow) decoding process

Hufmann encoding

- Coding algorithm can rely on a reasonable small Look-Up Table (LUT)
  - For a 5-symbol alphabet: 3-input LUT with 4 outputs
    * This is a 32-bit memory
  - For a 128-symbol alphabet: 7-input LUT with 127 outputs
    * This is a 2KB memory

- A memory of 2KB should not be a problem even for an embedded system

- If the coding LUT is still too large for the considered embedded system
  - Subdivide the coding LUT into smaller LUTs and perform the coding process in several steps
  - Penalty: larger coding time

- What would a Huffman encoder implementation look like?
  - Huffman encoding does not pose difficult technical problems
  - Huffman decoding is a far more difficult task!

Possible Huffman encoder implementation strategies

- A single large LUT
  - The main code just access the LUT in order to retrieve the codeword
  - The LUT's word-width is equal to the longest codeword
- Several smaller LUTs
  - The LUT's word-width is smaller
  - The coding process is performed in several steps
- These strategies can be implemented both in:
  - Hardware: the LUT(s) are implemented within the functional unit
  - Software: the LUT(s) are stored into memory (ideally in cache)

Pure-software implementation of the Huffman encoder

```
#include <stdio.h>
char *HE_LUT[5] = { "0", "10", "110", "1110", "1111"};

int main( void) {
  char symbol_to_encode = 0;

  do {
    scanf( "%i", &symbol_to_encode);
    printf( "%s\n", HE_LUT[symbol_to_encode - 0x40]);
  } while ( (symbol_to_encode > 0x40) & (symbol_to_encode < 0x46));
  printf( "%s\n", "Not a valid symbol.");
  exit( 0);
}
```

- ASCII code of character 'A' is 0x41
- ASCII code of character 'E' is 0x45

Hufmann decoding

- A Hufmann-encoded string: 11010011101111010

```
110   10   0   1110   1111   0   10
  C    B   A      D      E   A    B
```

- To achieve maximum compression, the coded data does not contain specific guard bits separating consecutive codewords
- The decoding process must:
  - Determine the symbol itself
  - Determine the code-length of the symbol
  - Shift the incoming string in order to discard the decoded bits
- Before initiating a new decoding iteration, the input string has to be shifted by a number of bits equal to the decoded code-length
  - A new symbol cannot be decoded before the current one has been decoded
- There are a lot of recursive operations that generate true-dependencies

Hufmann decoding

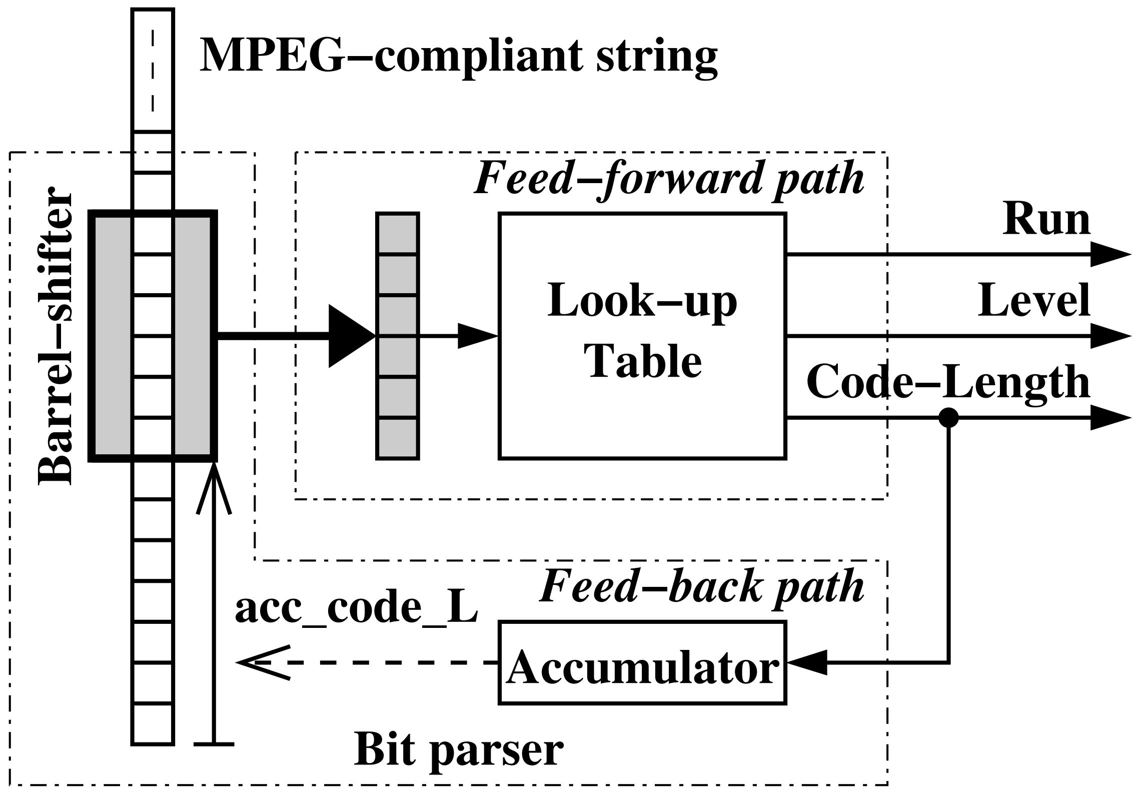
- Hufmann decoding is intrinsically a sequential process
- Parallel processing capabilities are not likely to improve the decoding rate
  - Pipelined engine
  - Horizontal engine
- Providing Huffman decoding hardware support is worth to be considered
- Will the processor be idle while the Huffman unit decodes the input string?
- Combine Huffman decoding with other tasks, for example:
  - Run-Length Decoding (RLD)
  - Inverse Discrete Cosine Transform (IDCT)

Hufmann decoding – the brute force approach

- Select a chunck of the incoming string that has a number of bits equal to the largest code-length
- Look-up into a Huffman decoding table with the selected chunck as address
- The LUT returns:
  - The bit combination of the decoded symbol
  - The code-length of the decoded symbol
- Discard *code-length* bits from the incoming string
- This approach is good for very small code-lengths since the LUT is small
- For large code-lengths the LUT size becomes very large!
  - MPEG: the longest codeword (excluding Escape!) is 17 bits $\longrightarrow$ the LUT size reaches $2^{17} = 128$ K words for a direct mapping of all possible codewords
  - MPEG: the symbol is a combination of a *run* code and a *length* code

University of Victoria
British Columbia • Canada

Huffman (variable-length) decoding principle



- VLD performance: the throughput is bounded by the inverse to the loop latency

University of Victoria
British Columbia • Canada

Huffman (variable-length) decoding principle

- VLD is a system with feedback, whose loop typically contains:
  - Look-Up Table on the feed-forward path
  - Bit parser on the feedback path
- LUT receives the variable-length code itself as an address and outputs:
  - the decoded symbol (*run-level* pair or *end_of_block*)
  - the codeword length
- To determine the starting position of the next codeword, the *code_length* is fed back to an accumulator and added to the previous sum of codeword lengths,
- The bit parsing operation is completed by the *barrel-shifter* (or *funnel-shifter*) which shifts out the decoded bits.

University of Victoria
British Columbia • Canada

Huffman (variable-length) decoding performance

- The throughput is bounded by the inverse of the loop latency

- Major goal: reduce the loop latency!
  - Reduce the operation budget
    - Look-up operation
    - Accumulation
    - Barrel-shifting
  - Reduce the latency of each operation

- Hardware issues regarding VLD parts
  - Barrel-shifter is essentially a DEMUX – implemented within the standard instruction set (that is, in software)
  - Adder that performs the accumulation should be high-performance (carry look-ahead, carry select, etc.)
  - LUT: low latency is more important than silicon area

Huffman decoding: reducing the operation budget

- Keep the accumulator out of the critical path:

  M.-T. Sun, *VLSI architecture and Implementation of a High-Speed Entropy Decoder*, Proceedings of the IEEE International Symposium on Circuits and Systems, 1991, pp. 200-203.

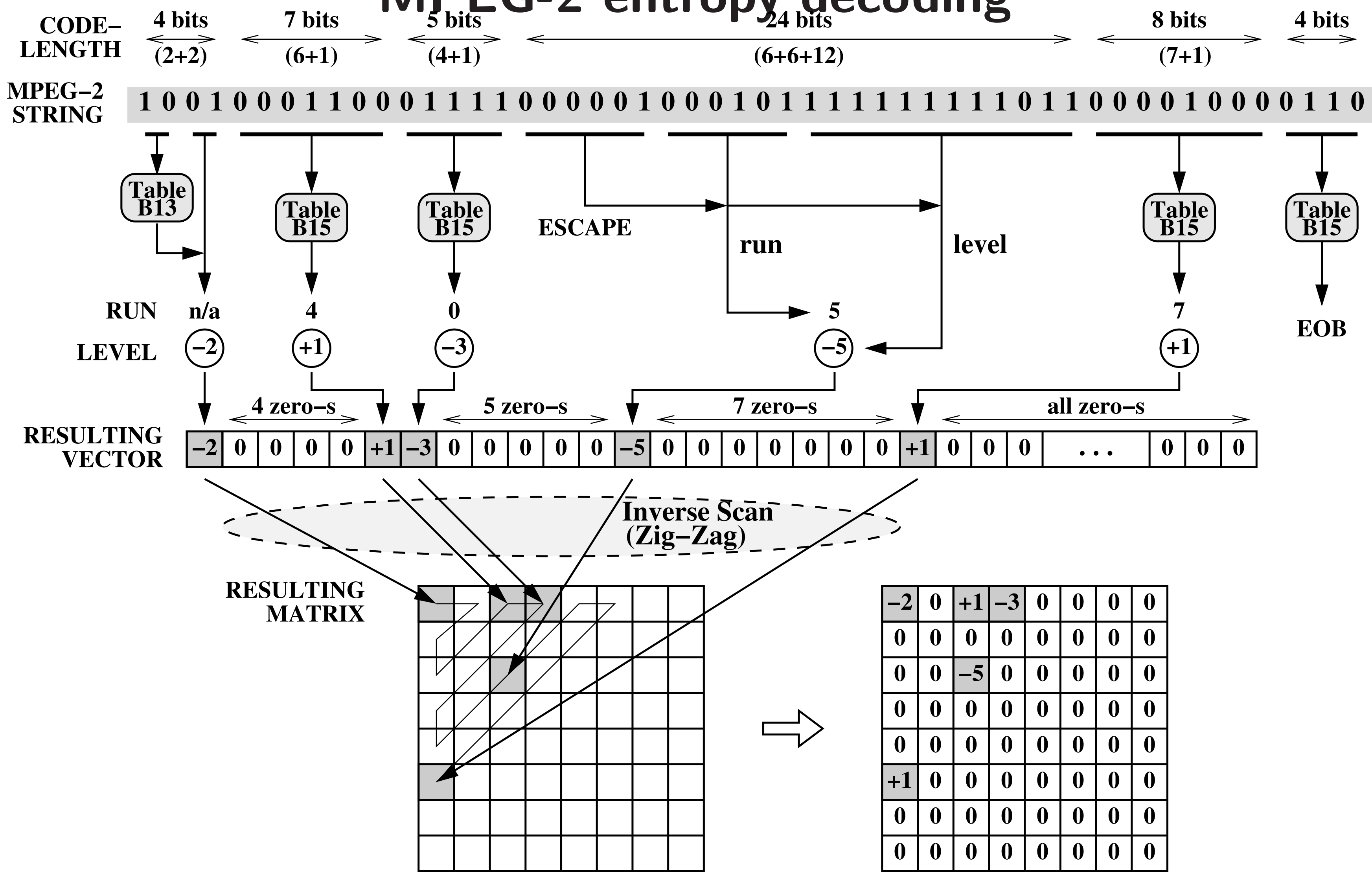
- Is multiple-symbol decoding possible?
  - What is really important is to detect the code-lengths to be able to initiate the next decoding iteration
  - What would be the LUT size in this case? Try multiple-symbol decoding for short codewords and single symbol decoding for long codewords.

- Try to split the accumulation operation is plain addition and storage

MPEG: Entropy decoding

- MPEG video coding standard:
  - DCT + Quantization: lossy compression
  - Entropy coding: lossless compression
- Entropy decoding consists of two distinct steps:
  - Variable-Length (Huffman) Decoding (VLD)
  - Run-Length Decoding (RLD)
- Both VLD and RLD are sequential tasks (due to data dependencies)
- Entropy decoding is an intricate function on parallel computing engines
- Entropy decoding is an ideal candidate to benefit from hardware support.

MPEG-2 entropy decoding



Hufmann decoding – project requirements

- Define your own alphabet
- Assume a particular distribution for the probabilities of occurence
- Define the Huffman codes and calculate the average transmission rate with and without Huffman coding
- Build the testbench (= a file that contains alphabet symbols occuring with the assumed probabilities)
- Provide a pure-software solution for Huffman decoding
  - Try to reduce the cache misses (do not use very large LUTs)
  - Estimate the performance for the particular testbench
- Try also a firmware solution, but since Huffman decoding is a sequential process do not expect any improvement

Hufmann decoding – project requirements

- Build a full-custom hardware unit for the Huffman decoder and estimate its performance against 32-bit addition
  - Reentrant or non-reentrant functional unit?
- Define a new instruction that will call the full-custom Huffman decoder
  - You must comply with the ARM architecture (you can have at most two arguments and one result per instruction call)
- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- Estimate the performance of the ARM processor augmented with a Huffman decoding unit
- Estimate the speed-up (if any) and the penalty in terms of number of gates required to implement the Huffman decoder

Questions, feedbacks

?