ECE 30
Introduction to Computer Engineering
Programming Project: Spring 2024
Encoding using Shannon–Fano tree
Due by Thursday, June 6 at 11:00am
(before last lecture)

Project TA: Junyi Xu

1 Project Description

In this project, you will be given an array of symbols with corresponding frequencies (i.e. rate of occurrence). The symbol array is already sorted by the frequencies in ascending order. Write a program to build a binary tree using the Shannon-Fano strategy and use the binary tree to encode a string consisting of these symbols. Notice that the binary tree generates an instantaneous and uniquely decodable code word for each symbol. Although the typical Shannon-Fano strategy can create different binary tree structures with the same symbol array depending on how the symbol array is split, we will use the *FindMidpoint* function to ensure that a unique tree structure will be obtained in response to the sorted symbol array. The detailed principle of the Shannon-Fano strategy is discussed below.

2 Shannon–Fano encoding algorithm

2.1 Building binary tree

The Shannon-Fano encoding strategy strives to reduce the number of bits used to store a list of symbols by assigning fewer bits to symbols that occur more frequently in a text. We need to place symbols with higher frequencies at an equal or lower level in the binary tree (closer to the root) than symbols with lower frequencies.

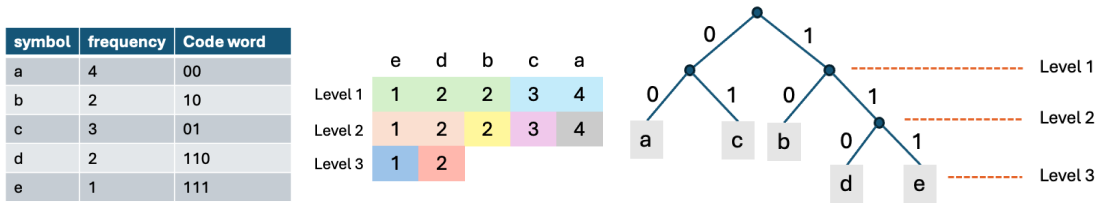
Then, we find the point of separation that splits the list into two parts with roughly equal total frequencies. For each part, we again look for the point of separation that divides the sub-array into two segments with roughly equal total frequencies. This process is repeated until the remaining part is indivisible.

For example, given a string *abecacadbcad*, we identify 5 unique characters and their associated frequencies shown in Figure 1a. The procedures to build a binary tree are listed as follows:

- Step 1: Sort the symbol array in ascending order based on their frequencies $\rightarrow \{e, d, b, c, a\}$
- Step 2: Split the sorted array into left part $\{c, a\}$ and right part $\{e, d, b\}$, where the left part has a total frequency of 7 and the right part has a total frequency of 5
- Step 3: Split the left part $\{c, a\}$ into $\{a\}$ and $\{c\}$, reaching the end of this part of array
- Step 4: Split the right part $\{e, d, b\}$ into $\{b\}$ and $\{e, d\}$, where $\{b\}$ has a frequency of 2 and $\{e, d\}$ has a frequency of 3
- Step 5: Split the left part $\{e, d\}$ into $\{e\}$ and $\{d\}$, reaching the end of this part of array

Figure 1b highlights the sets at different levels. We can convert this splitting procedure into a binary tree. At level 1, there is no leaf since both sets contain more than one symbol. At level 2, we have $\{a\}$, $\{b\}$, and $\{c\}$ containing only one symbol, so they are represented as leaf nodes.

At level 3, the remaining $\{d\}$ and $\{e\}$ become leaf nodes. The resulting binary tree is shown in Figure 1c. This will be implemented by *Partition* function recursively.



(a) (b) (c)

Figure 1: Computing the code word of each symbol in the string *abecacadbcad* using binary tree

To locate the most optimal splitting point, we use another recursive function *FindMidpoint* to trace from two sides of the array. We define two pointers **head pointer** and **tail pointer** pointing to the start and end of the symbol array respectively. The rules are listed as follows:

- If the sum of frequency to the **left of and including the head pointer** is **smaller than or equal to** the sum of frequency to the **right of and including the tail pointer**, move the **head pointer** to the **right** by one position
- If the sum of frequency to the **left of and including the head pointer** is **greater than** the sum of frequency to the **right of and including the tail pointer**, move the **tail pointer** to the **left** by one position
- If the **head pointer** and **tail pointer** point to two consecutive symbols, we stop tracing. The **head pointer** now points to the last symbol of the sub-array at the **left-hand side**. The **tail pointer** now points to the first symbol of the sub-array at the **right-hand side**.

Figure 2 illustrates how we use the set of rules to split $\{e, d, b, c, a\}$ in the previous example into $\{e, d, b\}$ and $\{c, a\}$

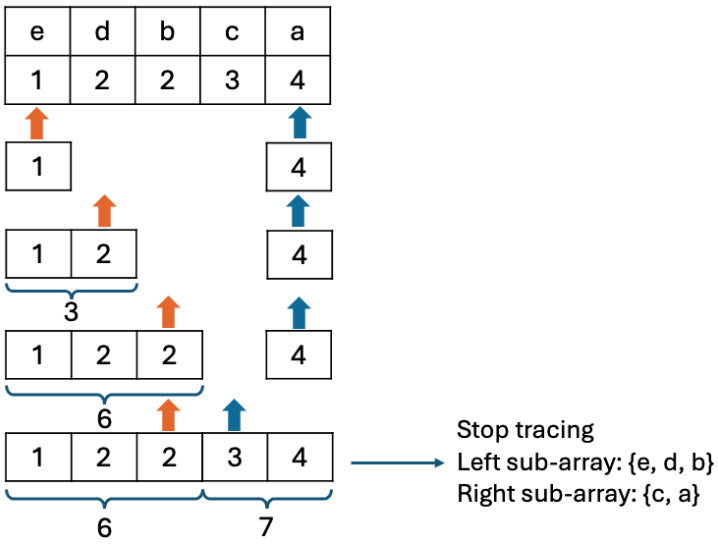


Figure 2: Example of finding the splitting point. The orange arrow represents head pointer and the blue arrow represents tail pointer.

2.2 Generate code word

With the binary tree, we can generate the code word for each symbol. For example, to encode d, we start by searching the symbols that belong to the left sub-tree of the root node. Since d does not belong to a, c, we proceed to the right node and search for the left sub-tree. Since d is not contained in b, we proceed to the right node and search for the left sub-tree again. Fortunately, d is directly found as the left leaf node. The code word of d is obtained by tracing the node. In this case, the code word for d is 110. Following the same procedure, we can compute the code words for all symbols, as shown in Figure 1a. Subsequently, we can encode the string *abecacadbcad* into 001011101000100110100100110.

2.3 Data structure for symbols and frequencies

The program stores an array of symbols and their frequencies in a series of consecutive memory spaces. Each symbol is followed by its frequency. The symbol array in Figure 3 is an example. All symbols will be represented using non-negative numbers. The end of the list is marked by a memory space storing the value -1. **For the simplicity of this project, all given symbol arrays will be sorted arrays based on the frequency of each symbol in ascending order.** In other words, there is no need to sort the given symbol array in this project.

2.4 Data structure for binary tree

To facilitate the encoding procedure, the binary tree structure used is slightly more complicated. Each node of a binary tree has four attributes: **start pointer**, **end pointer**, **left node**, and **right node**. **start pointer** and **end pointer** are used to identify the range of symbols in the sub-tree of a node in the given symbol array. Here are the detailed descriptions of each attribute:

- **start pointer**: the address that stores the first symbol of the symbol array section that represents the sub-tree
- **end pointer**: the address that stores the last symbol of the symbol array section that represents the sub-tree.
- **left node**: the address of the first attribute of the left node in the tree array.
- **right node**: the address of the second attribute of the right node in the tree array.

Figure 3 visualizes the tree array in a more detailed perspective. Node 1 is the root of the binary tree, so the start pointer and end pointer should include the whole symbol array. To represent this, the **start pointer** points to Symbol 1 in the symbol array and **end pointer** points to Symbol 2 in the symbol array. Its left node is Node 2, so **left node** points to the first memory slot corresponding to Node 2. Similarly, **right node** points to the first memory slot corresponding to Symbol 2. Node 2 covers a narrower range of symbols that only contains Symbol 1 and 3, so the **start pointer** points to 1 in the symbol array and **end pointer** points to 3 in the symbol array. For the leaf node corresponding to Symbol 2, both the **start pointer** and the **end pointer** will point to Symbol 2 in the symbol array. Since it is a leaf node with no child node, both **left node** and **right node** will store **NULL**. **In this project, NULL will be represented using -1.**

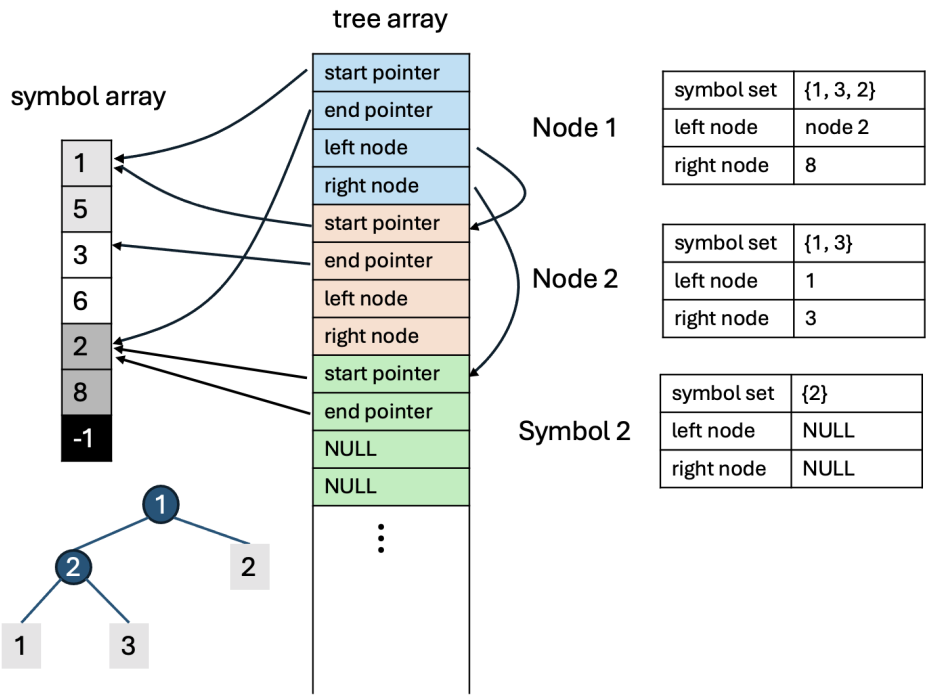


Figure 3: Data structure for symbol array and binary tree array. In the symbol array, the first number in the consecutive pair with the same color is the symbol and the second number is the frequency of that symbol. In the tree array, four consecutive memory spaces filled with the same color represent one tree node.

3 Implementation

The whole project needs 5 functions to implement the Shannon-Fano strategy in the LEGv8 assembly language: ***FindTail***, ***FindMidpoint***, ***Partition***, ***IsContain***, and ***Encode***. Some details to note:

- Use the procedures prototype as mentioned below and given to you in the template. Don't change the registers or the arguments passed to the procedures or the values returned.
- Follow the "Procedure Call Convention" for calling procedures, passing registers and managing the stack. The procedures should not make any assumptions about the implementation of other procedures, except for the name of input/output registers and the conventions mentioned in the course.
- We expect your code to be well-commented. Each instruction should be commented with a meaningful description of the operation. For example, this comment is bad as it tells you nothing:

```
// x1 gets x2 - 1
sub x1, x2, #1
```

A good comment should explain the meaning behind the instruction. A better example would be:

```
// Initialize loop counter x1 to n-1
sub x1, x2, #1
```

3.1 Function 1: FindTail(pt)

Find the last symbol of the symbol array

3.1.1 Parameters

- x0: The address of (pointer to) the first symbol of the symbol array (correspond to pt)

3.1.2 Return Value

- x1: The address of (pointer to) the last symbol of the symbol array

3.1.3 Pseudo-code

$*pt : symbol$
$*(pt + 1) : frequency$
$*(pt + 2) : next\ symbol$

function FINDTAIL(pt)
    if *(pt+2) = -1 **then**
        return pt
    else
        $pt \leftarrow pt + 2$
        return FINDTAIL(pt)
    end if
end function

3.1.4 Examples

symbol_array: 1 3 4 4 5 6 2 8 -1
Input:

- pt: address of *symbol_array*[0]

Output:

- address of *symbol_array*[6]

3.2 Function 2: FindMidpoint($head, tail, left\_sum, right\_sum$)

Split the symbol array into two sub-arrays of roughly the same total frequency and return the first element of the right-hand side sub-array.

3.2.1 Parameters

- x0: The address of (pointer to) the first symbol of the symbol array (correspond to *head*)
- x1: The address of (pointer to) the last symbol of the symbol array (correspond to *tail*)
- x2: The sum of the frequency of the left sub-array (correspond to $left\_sum$)
- x3: The sum of the frequency of the right sub-array (correspond to $right\_sum$)

3.2.2 Return Value

- x4: The address of (pointer to) the first element of the right-hand side sub-array

3.2.3 Pseudo-code

function FindMidpoint($head, tail, left\_sum, right\_sum$)
  if $head + 2 == tail$ **then**
    return tail;
  else
    if $left\_sum \leq right\_sum$ **then**
      $head \leftarrow head + 2$
      $left\_sum \leftarrow left\_sum + *(head + 1)$
    else
      $end \leftarrow end - 2$
      $right\_sum \leftarrow right\_sum + *(tail + 1)$
    end if
    return FindMidpoint($head, tail, left\_sum, right\_sum$)
  end if
end function

3.2.4 Examples

1.
symbol_array: 3, 1, 1, 2, 5, 3, 7, 6, -1
Inputs:

- start: pointer to *symbol_array*[0]
- end: pointer to *symbol_array*[6]
- $left\_sum$: 1
- $right\_sum$: 6

Outputs:

- pointer to *symbol_array*[6]

2.
symbol_array: 3, 4, 2, 5, 1, 9, 7, 10, 4, 15, 20, 20, -1
Inputs:

- start: pointer to $symbol\_array[0]$
- end: pointer to $symbol\_array[8]$
- $left\_sum$: 4
- $right\_sum$: 20

Outputs:

- pointer to $symbol\_array[8]$

3.3 Function 3: Partition($start, end, node$)

Build a binary tree by recursively splitting the symbol array.

3.3.1 Parameters

- x0: The address of (pointer to) the first symbol of the symbol array (correspond to *start*)
- x1: The address of (pointer to) the last symbol of the symbol array (correspond to *end*)
- x2: The address of the first attribute of the current binary tree node (correspond to *node*)

3.3.2 Return Value

- This function does not return anything. It directly stores value to the memory space

3.3.3 Pseudo-code

$*node : node \rightarrow start$
$*(node + 1) : node \rightarrow end$
$*(node + 2) : node \rightarrow left$
$*(node + 3) : node \rightarrow right$

function PARTITION(start, end, node)
    $*node \leftarrow start$
    $*(node + 1) \leftarrow end$
    if $start = end$ **then**
        $*(node + 2) \leftarrow NULL$
        $*(node + 3) \leftarrow NULL$
    else
        $left\_sum \leftarrow *(start + 1)$
        $right\_sum \leftarrow *(end + 1)$
        $midpoint \leftarrow$ FINDMIDPOINT$(start, end, left_sum, right_sum)$
        $offset \leftarrow midpoint - start - 1$
        $left\_node \leftarrow node + 4$
        $right\_node \leftarrow node + 4 + offset \times 4$
        $*(node + 2) \leftarrow left\_node$
        $*(node + 3) \leftarrow right\_node$
        PARTITION$(start, midpoint - 2, left_node)$
        PARTITION$(midpoint, end, right_node)$
    end if
end function

3.3.4 Examples

1.
symbol_array: 1 3 2 5 3 8 -1
Inputs:

- start: $symbol\_array[0]$
- end: $symbol\_array[4]$

- node: 7

When exiting:

- *tree_array*: 0 4 11 23 0 2 15 19 0 0 -1 -1 2 2 -1 -1 4 4 -1 -1

2.
symbol_array: array: 1 3 2 3 3 4 4 5 -1
Inputs:

- start: *symbol_array*[0]
- end: *symbol_array*[6]
- node: 9

When exiting:

- *tree_array*: 0 6 13 25 0 2 17 21 0 0 -1 -1 2 2 -1 -1 4 6 29 33 4 4 -1 -1 6 6 -1 -1

3.4 Function 4: IsContain($start, end, symbol$)

Check whether the sub-tree of the current node contains the symbol.

3.4.1 Parameters

- x0: The address of (pointer to) the first symbol of the sub-array (corresponding to *start*)
- x1: The address of (pointer to) the last symbol of the sub-array (corresponding to *end*)
- x2: The symbol to look for (corresponding to *symbol*)

3.4.2 Return Value

- x3: 1 if symbol is found, 0 otherwise

3.4.3 Pseudo-code

function IsContain($start, end, symbol$)
  while $start \leq end$ **do**
    if $*start = symbol$ **then**
      return 1
    end if
    $start \leftarrow start + 2$
  end while
  return 0
end function

3.4.4 Examples

1.
symbol_array: 1, 3, 3, 5, 2, 6, 4, 7, -1
Inputs:

- start: *symbol_array*[0]
- end: *symbol_array*[6]
- symbol: 3

Outputs:

- Output: 1

2.
symbol_array: 1, 3, 3, 5, 2, 6, 4, 7, -1
Inputs:

- start: *symbol_array*[0]
- end: *symbol_array*[4]
- symbol: 4

Outputs:

- Output: 0

3.5 Function 5: Encode($node, symbol$)

Print the code word of the given symbol

3.5.1 Parameters

- x0: The address of (pointer to) the binary tree node (correspond to *node*)
- x2: The symbol to encode (corresponding to *symbol*)

3.5.2 Return Value

- This function does not return anything.

3.5.3 Pseudo-code

$*left\_node : left\_node \rightarrow start$
$*(left\_node + 1) : left\_node \rightarrow end$

function ENCODE(node, symbol)
    $left\_node \leftarrow *(node + 2)$
    $right\_node \leftarrow *(node + 3)$
    if $left\_node \neq right\_node$ **then**
        if ISCONTAIN($*left\_node, *(left\_node + 1), symbol$) **then**
            print 0
            ENCODE($left\_node, symbol$)
        else
            print 1
            ENCODE($right\_node, symbol$)
        end if
    end if
end function

3.5.4 Examples

1.
symbol_array: 1 3 2 5 6 9 8 10 -1
tree_array: 0 6 13 33 0 4 17 29 0 2 21 25 0 0 -1 -1 2 2 -1 -1 4 4 -1 -1 6 6 -1 -1
Inputs:

- node: 9
- symbol: 2

Outputs:

- printed text: 001

2.
symbol_array: 1 2 2 3 3 3 4 4 -1
tree_array: 0 6 13 25 0 2 17 21 0 0 -1 -1 2 2 -1 -1 4 6 29 33 4 4 -1 -1 6 6 -1 -1
Inputs:

- node: 9

- symbol: 3

Outputs:

- printed text: 10

3.6 Main function

Encode all given symbols using the binary tree. (This function is the main body of the program, the code is already given to you).

3.6.1 Input from data file

- Array of symbols to be encoded
- Array of unique symbols and corresponding frequency

3.6.2 Examples

1.
Inputs:

- *symbol_array*: 1 3 2 5 6 9 8 10 -1
- *text_array*: 1 2 6 8 1 6

Outputs:

- printed text: 00000101100001

2.
Inputs:

- *symbol_array*: 1 2 2 3 3 3 4 4 -1
- *text_array*: 1 4 2 3 4 3 2

Outputs:

- printed text: 00110110111001

4 Additional comments

- The program will load a symbol array and a text array. The text array is guaranteed to consist only of the symbols given in the symbol array.
- The provided main function **starts the tree array immediately following the symbol array**. The address of the root node may be changed during grading, so do not hardcode the tree array.
- Two example data files are provided for reference. **test_partition .txt** is the example data file for testing the *Partition* algorithm. It only contains a symbol array. The other example data file, **test_encode.txt**, contains both a symbol array and a list of symbols to be encoded. This can be used for testing the whole program.
- The examples given after each function description assume that the start of the symbol array is stored at the 0th position of the memory space. In actual programming, since we need to pass the text array to the program, the storing position of each symbol and tree node will be shifted, which results in different tree arrays. Be sure to test your partition algorithm using the $test\_partition.txt$ separately before combining it with the encoding part.

- There is no MULI/DIVI operation, however LSL x0, x1, #3 achieves $x0 \leftarrow x1 \times 3$. Similarly, LSR x0, x1, #1 achieves $x0 \leftarrow x1/3$ as long as x1 is non-negative.
- You may find it helpful to manually work out the code word of each symbol according to the Shannon-Fano strategy during the debugging process, as it is more straightforward than the *tree_array* representation.

5 Instructions

- You must submit your solution on Canvas in a completed file ABC_XYZ_2021_project.s, where ABC and XYZ are the @ucsd.edu of each student. For example, if Sherlock Holmes and John Watson were working together on a submission, the file name would be: sholmes_jwatson_2024_project.s
- Fill the names and PID of each student in the file.
- Each project team contributes their own unique code – no copying, cheating, or hiring help. We will check the programs against each other using automated tools. These tools are VERY effective, and you WILL get caught!
- Please start this project early!
- Try to test the behavior of each function independently, rather than trying to code all of them at once. It will make debugging far easier.

6 Grading

- FindTail (2 points): 4 test cases, 0.5 point each
- FindMidpoint (3 points): 6 test cases, 0.5 point each
- Partition (3 points): 6 test cases, 0.5 point each
- IsContain (2 points): 4 test cases, 0.5 point each
- Encode (3 points): 6 test cases, 0.5 point each
- Extensive testing (2 points): points deducted if your code does not compile, run, or terminate correctly

Note: Your comments will not be graded, but your TA may refuse to proofread your code if your code is not well-commented.