CS5120 VLSI Final Project Report

Huffman coding Accelerator

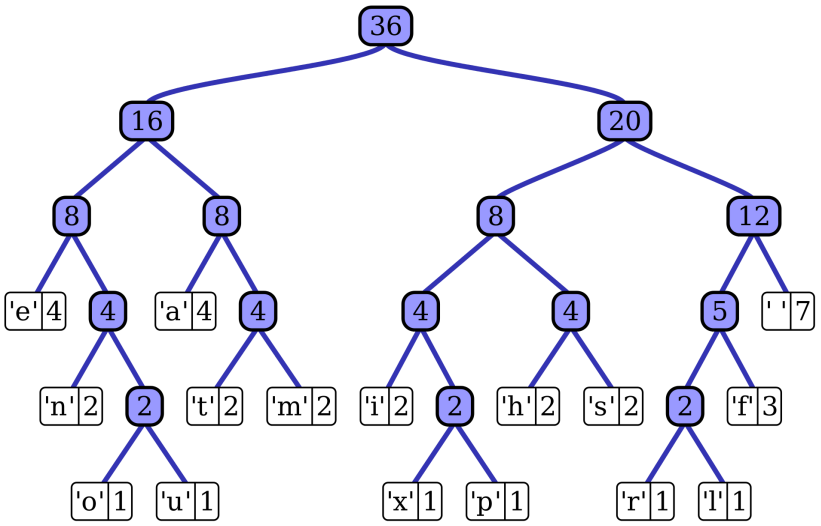
TEAM 23
106072139李禹承
107062107王領崧

Outline

- Introduction & Motivation
- Hardware Improvement
- Encode Architecture
- Decode Architecture
- Result
- Discussion
- Conclusion
- Contribution List
- Reference

Introduction & Motivation

Huffman coding 是一種很常見的無失真壓縮方式, 在現今的文件壓縮、影片(像)壓縮中, 大多也都是以 Huffman coding 去做延伸以及改良, 而最基礎的 Huffman coding 其實十分簡單易懂, 實作也很容易, 只需要先統計每個字符出現的頻率表後, 就可以依此去 construct tree 以及 encode symbol, 常見的 encode /decode 方式就是掃過整個 sequence 一次, 然後根據那個 table 來做 encode / decode 的動作, 然而只要有了 encode/decode 的轉換 table 後, 其實各個 symbol 間的 decode / encode 其實是可以獨立運行的, 並不需要依照順序執行過去, 因此這讓我們有個想法, 想結合上課所學, 利用硬體來加速 huffman encoding / decoding 的執行。



字母	頻率	編碼
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

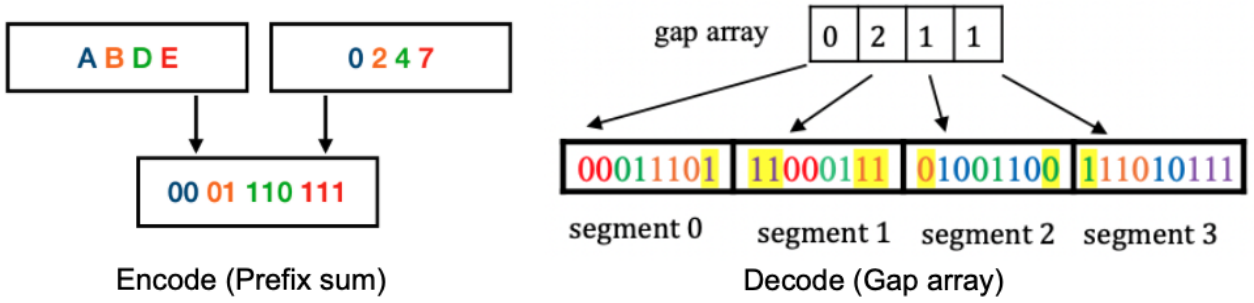
Hardware improvement

- Overall
  細看 Huffman coding 會發現，當有了 encode/decode 的 lookup table 後，只要我們能有 position table 後，其實每個 symbol 和每段特定的 encoded sequence 是可以獨立執行，因此我們為 encoder 創建一個 prefix sum，decoder 創建一個 gap array，讓執行平行化，底下會分成兩部份來詳細說明

- Prefix Sum
  它用來執行Encode的硬體加速。因為我們要平行的放這些 symbol 的 encoded sequence，就必須知道每一個symbol開始的位置，所以會計算出一個symbol之前的所有長度稱為prefix sum，也就是我們到時候要放置該symbol encode sequence的起始位置。一個encoder裡面會有三個symbol等待編碼，會先計算出每一個encoder裡面三個symbol的總長度，接著用總長度依序減掉前一個symbol的長度，就可以得到該symbol的prefix sum來當作我們到時候放置編碼的起始位置。

- Gap Array
  一般的Huffman decoding是依照順序將encode sequence去對建立好的table查表，並把相對應的symbol找出來並解碼。而我們想要平行進行解碼，故需要從encode sequence中去找到一個symbol的左右邊界，這樣才可以同時進行。在上方encode階段，因為會將 encode sequence 以每 8 個 bit 存放在一個 segment 裡面，所以當該 symbol 的 prefix sum 超過 8 的倍數的話，就代表它的 encode suquence會被截斷，會有跨 segment 存放的問題，所以用 gap array 來記錄每段 segment 的左右邊界。在平行解碼當中，我們會利用查看 Gap array 來決定的左右邊界，這樣 decoder 就可以直接從左界開始 decode 到右界，彼此互不干擾，也不會產生因為 decode 起始位置有誤而導致無法 decode 的錯誤發生。

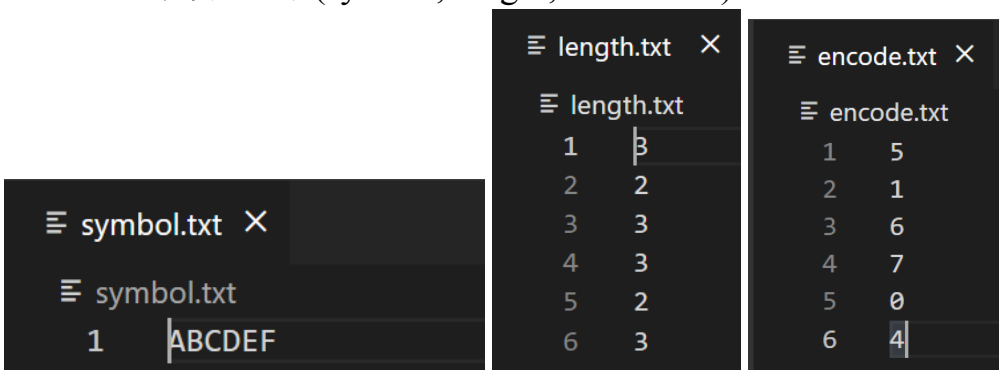
  底下為 encode / decode 簡單的示意圖



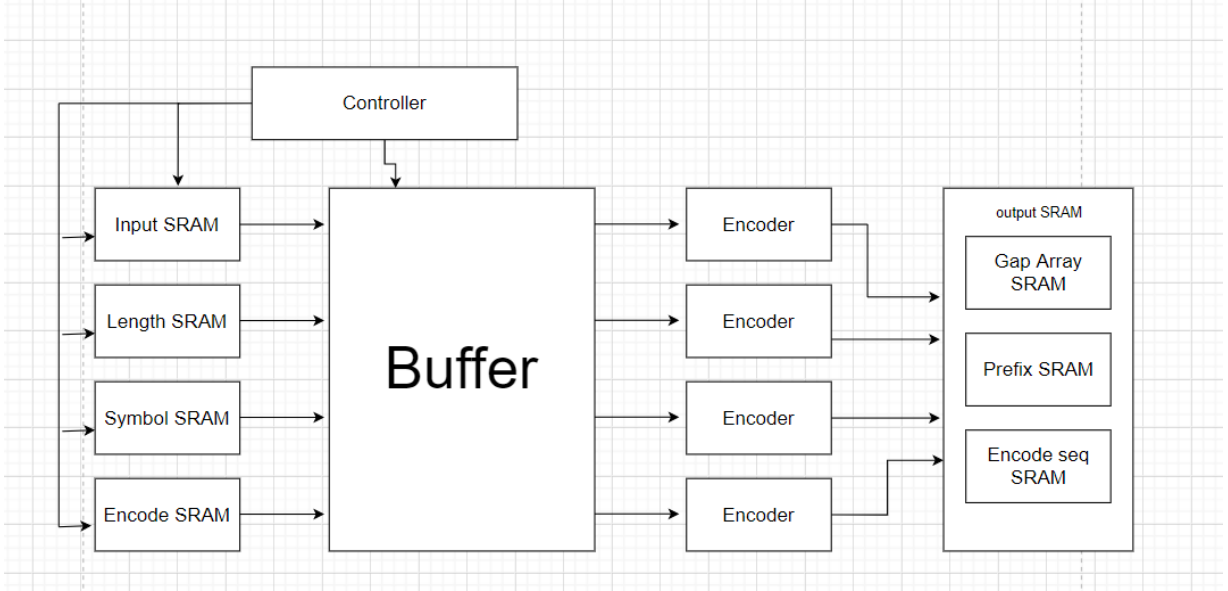
Architecture

[Encode]
一開始我們會利用軟體產生 encode input sequence 所需要的 symbol.txt, length.txt, encode.txt。
symbol.txt -> 這個Input sequence裡面有什麼symbol
length.txt -> 該symbol編碼完的長度
encode.txt -> 該symbol的編碼
(這邊會用Decimal來表示, 在encoder裡面會直接轉成Binary來進行運算)

底下左至右分別為 (symbol, length, encode.txt)



接著會從sram把這些資料都讀進來存在buffer, 透過symbol.txt & length.txt可以算出每個symbol的prefix sum, 這個prefix sum就會是我們編碼完之後要放置的位置。
然後有一個controller來控制資料的流動, 來把prefix sum & encode sequence 丟進encoder來進行編碼的動作。
這邊會利用四個encoder同時進行編碼, 一次會有四個symbol完成編碼並放到相對應位置的output SRAM, 而一個encoder裡面會有三個symbol等待編碼, 因此經過一輪運算之後, 總共會有12個symbol完成編碼。



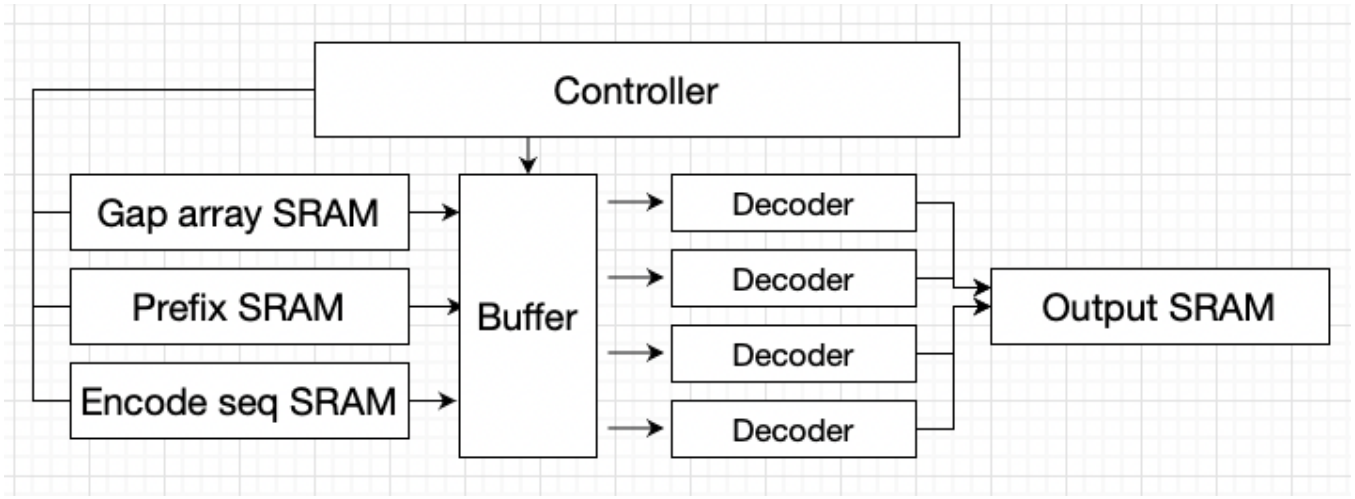
Architecture

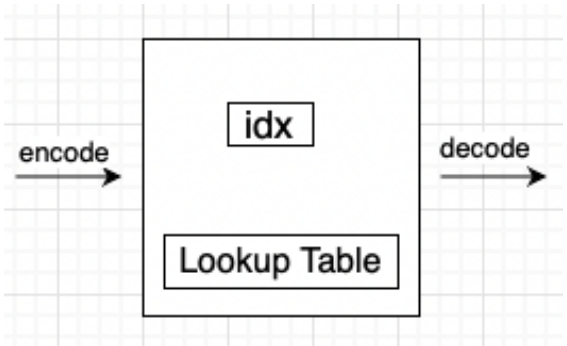
[Decode]

Decode 會根據 Encoder 傳進來的 gap array (代表 segment 的左右界)、encode sequence (編碼後的完整字串)、prefix (代表每個 segment decode 後的 symbol 要從哪一個位置開始放置), 先將它讀取至 buffer 後, 由 controller 根據 decoder 的 status, 來決定是否要傳送資料進去。由於每個 segment 都是 8-bit, 因此與 SRAM 溝通的方式與 hw3 一樣, 一次傳送都是 32-bit (也就是4 個 segment), 來防止 4 個 decoder 同時都 decode 完成需要新的 sequence 的 worst case 發生。

Decoder 裡面為一個不斷更新 idx & 查看 lookup table 的機制，lookuptable 是一個由 array 建成的 tree, 會在 decode 的動作開始前被 load 進來, 在接下來的 decode 過程中, idx 會根據讀取的 symbol, *2 or *2+1 (用來 traverse array), 並且 access 對應位置的 lookup table, 當發現是一個 symbol 後, 表示解碼成功, 就會將它 output 出去, 由 controller 儲存至 output SRAM, 反之就繼續讀取 sequence 的 symbol, 持續執行直到沒有 sequence 被傳送進來。

由於 decoder module 與 SRAM 的互動與 hw3 非常相似, 因此轉換到 PicoRV32 上面執行時, 做法也與 hw4 一樣, 所以這邊就不多做解釋。





Results

[Encode]

此為encode的實驗結果, 我們利用生成的 output encode sequence 與 golden data 進行比較, 發現編碼出來的結果都是正確的。但是有一個要注意的地方是當初在設計 prefix sum的平行化流程時出了一點問題, 現在只能確保第一輪運算出的encode sequence是正確的, 在後面的幾輪的運算有可能會讓 encode sequence 擺放的 sram 位置出問題, 導致最後的 encode sequence 是錯誤的, 此情況在下方的討論有提出來。但在這邊可以看到透過我們的平行化設計, 還是可以在第一輪將正確的encode sequence 運算出來且是存在正確的SRAM位置。

```
>> Check the output data
[Success] golden[          0]=10101110 | data[          0]=10101110
[Success] golden[          1]=11100000 | data[          1]=11100000
[Success] golden[          2]=00000111 | data[          2]=00000111
[Success] golden[          3]=11111100 | data[          3]=11111100
```

[Decode]

decode 分為兩部分做測試, 第一部分使用之前 hw3 的做法, 將 decoder module 完成後, 利用 testbench 讀取 golden data 來比對正確性。等到確認無誤後, 再把他放到 PicoRV32 上面與純軟體進行比較。底下有個圖表統計單個 decoder 和 4 個 decoder (此次實作) 進行比較, 以及在 PicoRV32 上面進行的軟硬體比較。

1 個 decoder vs 4 個 decoder : 理論上來說，4 個 decoder 執行的速度應該要是 4 倍, 然而因為 segment 不等長、segment 個數不一定是 4 的倍數等原因, 沒辦法時時刻刻 4 個 decoder 都同時有在 decode sequence, 實際跑的結果大約都落在 2-3 倍間, 但可以發現隨著 encode sequence length 的增長, 相差的速度會加大, 我們認為很可能的原因是因為隨著長度的增加, decoder 要 decode 的 round 就為越來越多, 因此有更大的機率能去平衡掉彼此間速率的不對等, 或是讓速度相差至多整數倍的 round, 這樣就能大大降低有 decoder 產生 idle 的情況發生，進而增加平行化的效益。

```
>> Check the output data
[Success] golden[          0]=A | data[          0]=A
[Success] golden[          1]=B | data[          1]=B
[Success] golden[          2]=C | data[          2]=C
[Success] golden[          3]=D | data[          3]=D
[Success] golden[          4]=E | data[          4]=E
[Success] golden[          5]=E | data[          5]=E
[Success] golden[          6]=E | data[          6]=E
[Success] golden[          7]=E | data[          7]=E
[Success] golden[          8]=E | data[          8]=E
[Success] golden[          9]=D | data[          9]=D
[Success] golden[         10]=D | data[         10]=D
[Success] golden[         11]=D | data[         11]=D
[Success] golden[         12]=C | data[         12]=C
[Success] golden[         13]=C | data[         13]=C
[Success] golden[         14]=B | data[         14]=B
```

```
[Success] golden[         15]=F | data[         15]=F
[Success] golden[         16]=F | data[         16]=F
[Success] golden[         17]=B | data[         17]=B
[Success] golden[         18]=A | data[         18]=A
[Success] golden[         19]=B | data[         19]=B
[Success] golden[         20]=C | data[         20]=C
[Success] golden[         21]=B | data[         21]=B
[Success] golden[         22]=A | data[         22]=A
[Success] golden[         23]=B | data[         23]=B
[Success] golden[         24]=E | data[         24]=E
[Success] golden[         25]=B | data[         25]=B
[Success] golden[         26]=D | data[         26]=D
[Success] golden[         27]=A | data[         27]=A
[Success] golden[         28]=A | data[         28]=A
[Congratulation!] all value are correct!
[Time usage]       1450 ns
```

Encode length	O(n)	Decoder	PicoRV32 soft	PicoRV32 hard
72	72 (cycle)	36.25 (cycle)	22329	293
176	176 (cycle)	62.25 (cycle)	51765	385
360	360 (cycle)	119.25 (cycle)	123815	675

合成結果:顯示的 required time 為 39.93, 也就是 critical path 所需要的傳輸時間, 因為小於所設定的 clock cycle 40ns, 表示我的 design 符合設定的頻率, 不會因為 delay 產生數值亂掉的問題。

```
clock clk (rise edge)                    40.00      40.00
clock network delay (ideal)               0.00      40.00
remain_reg[1]/CK (DFFRHQX1)               0.00      40.00 r
library setup time                       -0.07      39.93
data required time                                  39.93
-----------------------------------------------------------
data required time                                  39.93
data arrival time                                   -3.40
-----------------------------------------------------------
slack (MET)                                         36.53
```

Discussion

1. Encoding 實作問題:我們在實作Encode部分時遇到了一些問題。一個encoder裡面會有三個smybol等待編碼, 而我們會有四個Encoder同時在進行編碼, 所以一次會有四個symbol得到相對應的encode sequence, 並且透過該symbol的prefix sum來決定最後output SRAM的擺放位置。然而如果在這一輪所有symbol都編碼完之後, 最後一個output SRAM若沒有剛好填滿8個bit的話, 那麼在下一輪開始, 每次symbol編碼出來所放置的位置可能會有誤, 會造成最後encode sequence錯誤。我們認為是一開始的prefix sum的平行化設計有缺失, 目前的設計是在每一輪encode當中, 會去平行的計算每個symbol的prefix sum, 然後到下一輪的時候, 這些prefix sum就被洗掉重新計算了, 應該要想辦法讓之後的每一輪就繼續存著之前的prefix sum, 這樣才可以在encode的最後階段計算出正確的output SRAM擺放位置。
2. encoded length 越長 decoder 加速效果越好:encode length較短的時候, 4 個 decoder 所 decode 的 round 都不多, 這樣的情況下, 容易使得彼此 decode 的速度差異浮現出來, 造成在接近 decode 完成時, 很可能發生不同步的完成, 使得平行的效益沒有最大化。相反地, 如果 encode length 拉長, 彼此的速度差異就不會那麼明顯 (或是說會差到整數 round)，使得同個時間幾乎所有的 decoder 都有在運作, 達到平行化的最大效益。

Conclusion

1. 實作出Huffman decoding的加速器 on PicoRV32
2. 利用 prefix sum 平行化 encoder 來提高執行速度
3. 利用 gap array, idx array 平行化 decoder 來提高執行速度
4. 4 個 decoder 結合起來的 Decoder, 會比只有單一 decoder 還要快 2-3 倍
5. 以 PicoRV32 為平台, 硬體執行速度比起軟體能快上 100-200 倍

Contribution List

李禹承 : 實作 Encode部分, PPT製作, 架構討論, 撰寫Report
王領崧 : 實作 Decode部分, PPT製作, 架構討論, 撰寫Report

Reference

https://jnamaral.github.io/icpp20/slides/Yamamoto_Huffman.pdf