Energy Efficient Canonical Huffman Encoding

Janarbek Matai\(^1\), Joo-Young Kim\(^2\) and \textit{Ryan Kastner}\(^1\)

\(^1\) Department of Computer Science and Engineering
University of California, San Diego

\(^2\) Microsoft Research
Motivation

- Power consumption accounts for over 35% of total cost of ownership of datacenter\(^1\)
- Trends toward energy efficiency
  - ARM\(^3\)
  - FPGA\(^4\)

\(^2\)http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/
\(^4\)Andrew Putnam et al. “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services,” ISCA, 2014
Microsoft Catapults geriatric Moore's Law from CERTAIN DEATH

FPGAs DOUBLE data center throughput despite puny power pump-up, we're told

By Jack Clark, 16 Jun 2014

Bridging the IT gap between rising business demands and ageing tools

Microsoft has found a way to massively increase the compute capabilities of its data centers, despite the fact that Moore's Law is wheezing towards its inevitable demise.

In a paper to be presented this week at the International Symposium on Computer Architecture (ISCA), titled *A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services*, a troupe of top Microsoft Research boffins explain how the company has dealt with the slowdown in single-core clock-rate improvements that has occurred over the past decade.

To get around this debilitating problem -- more on this later -- Microsoft has built a system it calls Catapult, which automatically offloads some of the advanced tech that powers its Bing search engine onto clusters of highly efficient, low-power FPGA chips attached to typical Intel Xeon server processors.
Contributions

• Data encoding is popular data center application
  • Canonical Huffman Encoding: Microsoft’s Xpress compression algorithm[5]
• Design a hardware accelerated version using HLS on an FPGA
• Understand trade-offs between FPGA, ARM, and high-end CPU

Canonical Huffman Encoding
Canonical Huffman Encoding

Filter → Sort → Create Tree → Compute Bit Len → Truncate Tree → Canonize → Create Codeword

<table>
<thead>
<tr>
<th>S</th>
<th>F</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Canonical Huffman Encoding

Filter
Sort
Create Tree
Compute Bit Len
Truncate Tree
Canonize
Create Codeword

Filter
Sort
Create Tree
Compute Bit Len
Truncate Tree
Canonize
Create Codeword

<table>
<thead>
<tr>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

Canonical Huffman Encoding

```
17

0

1

0

10

1

0

1

7

4

D

E

2

C

1

0

1

0

0

1

1

A

B

F
```
Canonical Huffman Encoding

Filter → Sort → Create Tree → Compute Bit Len → Truncate Tree → Canonize → Create Codeword

Filter:
- S  F
  - A 3
  - B 1
  - C 2
  - D 5
  - E 5
  - F 1
  - G 0

Sort:
- S  F
  - A 3
  - B 1
  - C 2
  - D 5
  - E 5
  - F 1

Create Tree:
- S  F
  - A 3
  - B 1
  - C 2
  - D 5
  - E 5
  - F 1

Compute Bit Len:
- L  N
  - 2 3
  - 3 1
  - 4 2

Truncate Tree:
- L = Length
- N = Number of Symbols with Length L

Create Codeword:

N = Number of Symbols with Length L
Canonical Huffman Encoding

Filter → Sort → Create Tree → Compute Bit Len → Truncate Tree → Canonize → Create Codeword

- S  F
  - A  3
  - B  1
  - C  2
  - D  5
  - E  5
  - F  1
  - G  0

- S  F
  - A  3
  - B  1
  - C  2
  - D  5
  - E  5
  - F  1

Create Tree:

```
  17
 / \  
 0   1
 /   / 
7   10
 /   / 
A   D   E
```

Compute Bit Len:

```
L    N
---  ---
2    3
3    1
4    2
```

Truncate Tree:

```
L    N
---  ---
2    3
3    1
4    2
```

Canonize:

```
L    N
---  ---
2    3
3    1
4    2
```

Create Codeword:

- L = Length
- N = Number of Symbols with Length L
Canonical Huffman Encoding

S  F
A  3
B  1
C  2
D  5
E  5
F  1
G  0

Filter
Sort
Create Tree
Compute Bit Len
Truncate Tree
Canonize
Create Codeword

L=Length
N = Number of Symbols with Length L
Canonical Huffman Encoding

Filter -> Sort -> Create Tree -> Compute Bit Len -> Truncate Tree -> Canonize -> Create Codeword

S  F
A  3
B  1
C  2
D  5
E  5
F  1
G  0

Create Tree:

Create Codeword:

L = Length
N = Number of Symbols with Length L
Canonical Huffman Encoding

Canonical Huffman Encoding is memory efficient
Canonical Huffman Decoding is simple
Canonical Huffman Encoding

Canonical Huffman Encoding is memory efficient
Canonical Huffman Decoding is simple
Radix Sort

123
2
999
609
111
# Radix Sort

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>999</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>609</td>
<td>6</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Radix Sort
### Radix Sort

<table>
<thead>
<tr>
<th></th>
<th>1 2 3</th>
<th>1 1 1</th>
<th>6 0 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0 0 2</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
<tr>
<td>9 9 9</td>
<td>1 2 3</td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>6 0 9</td>
<td>9 9 9</td>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>6 0 9</td>
<td>9 9 9</td>
<td></td>
</tr>
</tbody>
</table>
Radix Sort

<table>
<thead>
<tr>
<th>123</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>999</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>609</td>
<td>6</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>6</th>
<th>0</th>
<th>9</th>
<th>0</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

123 → 111 → 609 → 002 → 111 → 123 → 111 → 123 → 609 → 999 → 999
Radix Sort

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1 1 1</td>
<td>6 0 9</td>
</tr>
<tr>
<td>2</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
<tr>
<td>999</td>
<td>9 9 9</td>
<td>1 2 3</td>
</tr>
<tr>
<td>609</td>
<td>6 0 9</td>
<td>9 9 9</td>
</tr>
<tr>
<td>111</td>
<td>1 1 1</td>
<td>6 0 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 9 9</td>
</tr>
</tbody>
</table>

**Pipelined stages in hardware**

**Each “stage” is a “Counting sort”**

<table>
<thead>
<tr>
<th>Algorithm 1 Counting sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: /*k is number of different radix</td>
</tr>
<tr>
<td>2: /*N is input array size</td>
</tr>
<tr>
<td>3: HISTOGRAM-KEY:</td>
</tr>
<tr>
<td>4: for i ← 0 to k do</td>
</tr>
<tr>
<td>5: Bucket[i] ← 0</td>
</tr>
<tr>
<td>6: end for</td>
</tr>
<tr>
<td>7: for j ← 0 to N − 1 do</td>
</tr>
<tr>
<td>8: Bucket[A[j]] ← Bucket[A[j]] + 1</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: PREFIX-SUM:</td>
</tr>
<tr>
<td>12: First[0] ← 0</td>
</tr>
<tr>
<td>13: for i ← 1 to k do</td>
</tr>
<tr>
<td>14: First[i] ← Bucket[i − 1] + First[i − 1]</td>
</tr>
<tr>
<td>15: end for</td>
</tr>
<tr>
<td>16: COPY-OUTPUT:</td>
</tr>
<tr>
<td>17: for j ← 0 to N − 1 do</td>
</tr>
<tr>
<td>18: i ← A[j]</td>
</tr>
<tr>
<td>19: Out[First[i]] ← temp[j]</td>
</tr>
<tr>
<td>20: First[i] ← First[i] + 1</td>
</tr>
<tr>
<td>21: end for</td>
</tr>
</tbody>
</table>

**RAW: II > 1**

**Ideal: II=1**

To get II=1, eliminate RAW dependencies using a technique that uses an “Accumulate” register.
Radix Sort

Algorithm 1 Counting sort

1: //k is number of different radix
2: //N is input array size
3: HISTOGRAM-KEY:
4: for i ← 0 to k do
5:   Bucket[i] ← 0
6: end for
7: for j ← 0 to N−1 do
8:   Bucket[A[j]] ← Bucket[A[j]] + 1
10: end for
11: PREFIX-SUM:
12: First[0] ← 0
13: for i ← 1 to k do
14:   First[i] ← Bucket[i−1] + First[i−1]
15: end for
16: COPY-OUTPUT:
17: for j ← 0 to N−1 do
18:   i ← A[j]
19:   Out[First[i]] ← temp[j]
20:   First[i] ← First[i] + 1
21: end for

int Accum = 0;
First[0] = 0;
for(int i=1; i<k; i++) {
    #pragma HLS PIPELINE II=1
    Accum = Accum + Bucket[i];
    First[i] = Accum;
}

Achieves II=1
IDEA: If current value of $A[]$ is equal to old value of $A[]$, then accumulate on a "Accum" register.

Otherwise write and read from different (old and current) memory locations.
```c
int old_value = 0;
int Accum = 0;
int value = 0;

value = A[j];
if (old_value == value) {
    Accum = Accum + 1;
}
else {
    Bucket[old_value] = Accum;
    Accum = Bucket[value] + 1;
}
old_value = value;
```
Radix Sort

```
int old_value = 0;
int Accum = 0;
int value = 0;

value = A[j];
if (old_value == value) {
    Accum = Accum + 1;
}
else {
    Bucket[old_value] = Accum;
    Accum = Bucket[value] + 1;
}
old_value = value;
```

old_value and value address locations are different $\rightarrow$ RAW dependency can be ignored using #pragma dependence
Radix Sort

<table>
<thead>
<tr>
<th>123</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>609</td>
<td>6</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Algorithm 1 Counting sort

```c
int old_value = 0;
int Accum = 0;
int value = 0;

value = A[j];
if (old_value == value) {
    Accum = Accum + 1;
} else {
    Bucket[old_value] = Accum;
    Bucket[value] = Accum + 1;
}
old_value = value;
```

old_value and value address locations are different $\rightarrow$ RAW dependency can be ignored using #pragma dependence
Radix Sort

Algorithm 1 Counting sort
1: //k is number of different radix
2: //N is input array size
3: HISTOGRAM_KEY:
4: for i ← 0 to k do
5: Bucket[i] ← 0
6: end for
7: for j ← 0 to N−1 do
8: Bucket[A[j]] ← Bucket[A[j]] + 1
9: end for
10: PREFIX-SUM:
11: First[0] ← 0
12: for i ← 1 to k do
13: First[i] ← First[i−1] + First[i−1]
14: end for
15: COPY-OUTPUT:
16: for j ← 0 to N−1 do
17: i ← A[j]
18: Out[First[i]] ← temp[j]
19: First[i] ← First[i] + 1
20: end for

#HLS_DEPENDENCE var=Bucket RAW false
int old_value = 0;
int Accum = 0;
int value = 0;
for(int j=0; j<N; j++) {
    #pragma HLS PIPELINE II=1
    value = A[j];
    if(old_value == value) {
        Accum = Accum + 1;
    }
    else {
        Bucket[old_value] = Accum;
        Accum = Bucket[value] + 1;
    }
    old_value = value;
}
Efficient Huffman Tree Creation
# Efficient Huffman Tree Creation

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
</tbody>
</table>
Efficient Huffman Tree Creation
Efficient Huffman Tree Creation

Table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
  n1
 /  \
F   B
  1  1
```
Efficient Huffman Tree Creation
Efficient Huffman Tree Creation
Efficient Huffman Tree Creation

```
S | F
---|---
F | 1
B | 1
C | 2
A | 3
D | 5
E | 5

n1 2
n2 4
```

```
    n2
     4
   /   \
 n1 2  C 2
     /   \
  F 1  B 1
```
Efficient Huffman Tree Creation

<table>
<thead>
<tr>
<th>S</th>
<th>F</th>
<th>n1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
<td>n2</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A 3

n2 4

n1 2

F 1

B 1

C 2
Efficient Huffman Tree Creation
Efficient Huffman Tree Creation
Efficient Huffman Tree Creation

Sort
Create Tree

<table>
<thead>
<tr>
<th>S</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
</tbody>
</table>

Create Tree:

```
  n5
  /|
 /  |
/   |
 n3  n4
   /|
  /  |
 /   |
 A   n2
  /|
 /  |
 /   |
 n1  C
  /|
 /  |
 /   |
 F   B
```

n5: 17
n3: 7
n4: 10
n2: 4
n1: 2
F: 1
B: 1
A: 3
C: 2
D: 5
E: 5
Efficient storage of Huffman Tree allows parallel bit length calculation
Parallel Bit Length Calculation

- This storage (Parent Address, Left, Right) allows parallel bit length calculation
- Partitioning the Huffman Tree in horizontal direction at any point will result independent data in Parent Address, Left and Right
Hardware Designs

I. Baseline
II. Restructured Design
III. Latency Optimized Design
IV. Throughput Optimized Design
Throughput Optimized Design

```c
CanonicalHuffman(SF[SIZE], Code[SIZE]){
    #pragma dataflow
    Filter(...);
    Sort(...);
    CreateTree(...);
    ComputeBitLenLeft(...);
    ComputeBitLenRight(...);
    TruncateTree(...);
    Canonize(...);
    CreateCodeword(...);
}
```

Pipelined stages achieved by dataflow pragma
CanonicalHuffman(SF[SIZE], Code[SIZE]){
#pragma dataflow
Filter(...);
Sort(...);
CreateTree(...);

ComputeBitLenLeft(...);
ComputeBitLenRight(...);

TruncateTree(...);
Canonize(...);
CreateCodeword(...);
}
CanonicalHuffman(SF[SIZE], Code[SIZE]){
#pragma dataflow

Filter(SF_IN, SF_TEMP);
Sort(SF_TEMP, SF_SORT1, SF_SORT2);
CreateTree(...);

ComputeBitLenLeft(...);
ComputeBitLenRight(...);

TruncateTree(...);
Canonize(...);
CreateCodeword(...);
}
Experimental Results

❖ Tools
  ➢ Vivado Suite and Vivado HLS

❖ Software Implementations:
  ➢ Intel Core i7 (3.6 GHz)
  ➢ Dual ARM® Cortex™-A9 MPCore™ of Zynq chip

❖ Hardware implementation platform:
  ➢ Xilinx Zynq Chip
  ➢ Processor
    ✓ Dual ARM® Cortex™-A9 MPCore™
    ✓ Up to 667 MHz operation
    ✓ 512 MB DDR3 memory
  ➢ FPGA: xc7z020clg484-1

❖ Test data:
  ➢ Size: 256, 536 and 704. (LZ77 uses 256 and 704 for text)
**Software Implementations**

**ARM Implementations**

- **Naïve**
- **Baseline**
- **SO**
- **SO+CS**

**Core i7 Implementations**

- **Naïve**
- **Baseline**
- **SO**
- **SO+CS**

---

**SO**: Software optimization by hand

**SO+CS**: Implementation with –O3 on top of SO

*ARM performance is calculated by using number of clock cycles from CP15 monitoring unit*
Hardware Implementations

**Baseline:** Initial design  
**Restructured:** After Radix sort, efficient Huffman tree and parallel bit length calculation  
**Latency Optimized design:** A design that does not have task level parallelism  
**Throughput Optimized:** A design that has task level parallelism
Software vs. Hardware Implementation: Performance

Throughput: HLS vs. ARM vs. Core i7

- **256**: 15X/1.8X
- **536**: 11X/1.4X
- **704**: 13X/1.5X

- Throughput Optimized
- ARM Optimized
- Core i7 Optimized
Software vs. Hardware Implementation: Energy Efficiency

Energy Efficiency: HW vs. ARM vs. Core i7

Core i7: Intel power gadget tool is used to get real time power
FPGA and ARM: Zynq-7000 ap soc measuring zc702 power using standalone application
Conclusions

+ High performance and low energy

- FPGA: higher throughput and lower energy solution for compression
- HLS tools are efficient if you understand how to use them
- Sequential data center algorithm can be implemented efficiently with HLS on an FPGA

Backup slides
Conclusion and Future Work

- Implemented an end-to-end Canonical Huffman encoding on a Zynq FPGA, ARM and Intel i7
- Presented several design decisions (e.g., efficient Huffman tree creation) that will result in efficient hardware in HLS for this application
- Presented performance and power trade-offs for different designs on an FPGA, ARM and Intel i7.
- *Lessons learned:* Sequential data center algorithm can be implemented efficiently with HLS on an FPGA
- *Future work:* Extend Canonical Huffman encoding from text to media (JPEG)
Overview

- Canonical Huffman Encoding Algorithm
- Hardware Implementation
  - Radix sort
  - Huffman tree creation
  - Parallel bit length calculation
- Hardware Designs
- Experimental Results
- Conclusion and Future Work
Canonical Huffman Encoding

For(i=Max Length-1 down to 1) {
    start[i] = start[i+1]+Len[i+1]
}

$N = \text{Number of Symbols with Length } L$
$L = \text{Length}$
Canonical Huffman Encoding

001 = (0000 + 2) >> 1
Canonical Huffman Encoding

Filter → Sort → Create Tree → Compute Bit Len → Truncate Tree → Canonize → Create Codeword

S | F | A | 3 | B | 1 | C | 2 | D | 5 | E | 5 | F | 1 | G | 0

Create Tree:

Filter:
S | F | A | 3 | B | 1 | C | 2 | D | 5 | E | 5 | F | 1 | G | 0

Sort:
S | F | A | 3 | B | 1 | C | 2 | D | 5 | E | 5 | F | 1 | G | 0

Create Tree:
S | F | A | 3 | B | 1 | C | 2 | D | 5 | E | 5 | F | 1 | G | 0

Alphabet:
S | C.C | A | 01 | B | 0000 | C | 001 | D | 10 | E | 11 | F | 0001

Compute Bit Len:
L | N | 2 | 3 | 3 | 1 | 4 | 2

Truncate Tree:
L | N | 2 | 3 | 3 | 1 | 4 | 2

Canonize:
S | L | A | 2 | B | 4 | C | 3 | D | 2 | E | 2 | F | 4

Create Codeword:
S | C.C | A | 01 | B | 0000 | C | 001 | D | 10 | E | 11 | F | 0001

001 = (0000 + 2)>>1

01 = (001 + 1)>>1

N = Number of Symbols with Length L
L = Length

Table:
<table>
<thead>
<tr>
<th>S</th>
<th>C.C</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>B</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>C</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0001</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Bit Length Calculation

Sort → Create Tree → Compute Bit Len

<table>
<thead>
<tr>
<th>Parent Address</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>F</td>
<td>n1</td>
<td>A</td>
<td>D</td>
<td>n3</td>
</tr>
<tr>
<td>Right</td>
<td>B</td>
<td>C</td>
<td>n2</td>
<td>E</td>
<td>n4</td>
</tr>
</tbody>
</table>

Address

0 1 2 3 4

Tree:

- Parent Address: 1 2 4 4 0
- Left: F n1 A D n3
- Right: B C n2 E n4

56
### Radix Sort

#### Pipelined stages

1. **Stage 1**
   - 1 2 3
   - 0 0 2
   - 9 9 9
   - 6 0 9
   - 1 1 1

2. **Stage 2**
   - 1 1 1
   - 1 2 3
   - 9 9 9
   - 6 0 9
   - 6 0 9

3. **Stage 3**
   - 6 0 9
   - 0 0 2
   - 0 1 1
   - 1 2 3
   - 9 9 9

---

#### Algorithm 1: Counting Sort

```plaintext
Algorithm 1 Counting sort
1: HISTOGRAM-KEY:
2: for i ← 0 to 2^t - 1 do
3:   Bucket[i] ← 0
4: end for
5: for j ← 0 to N - 1 do
7: end for
8: PREFIX-SUM:
9:   First[0] ← 0
10:  for i ← 1 to 2^t - 1 do
11:    First[i] ← First[i - 1] + First[i - 1]
12: end for
13: COPY-OUTPUT:
14:   First[0] ← 0
15:  for j ← 0 to N - 1 do
16:    Out[i] ← First[i] + 1
17: end for
18: end for
```

#### Code Snippet

```plaintext
#pragma dependence var=Bucket RAW false
val = f[i]
if(old_value = value) {
    accu = accu +1;
}
else {
    Bucket[old_value] = accu;
    accu = Bukcet[val] +1;
}
old_value = value;
```