

ECE 734 – VLSI Array Structures for Digital Signal Processing
Project Report

**An FPGA Implementation of the Fast Minimum-
Redundancy Prefix Coding**

Chia-Hsiung (Eric) Chen

ECE 734 – VLSI Array Structures for Digital Signal Processing
Project Report

An FPGA Implementation of the Fast Minimum-Redundancy Prefix Coding

Chia-Hsiung (Eric) Chen

Abstract

Minimum-redundancy prefix coding is a low-complexity, high compression ratio, and lossless entropy coding scheme for satellite images. In this presentation, we would like to propose an FPGA implementation of the minimum-redundancy prefix coder. Detailed algorithm transform, loop transform, pipelining, and scheduling techniques are discussed. Preliminary result shows that the proposed design is able to reach a high worst-case throughput rate at 78.562 Msamples/s, which is 3.928 times better than the CCSDS Rice coding chip. With the low power consumption at 0.0105 W/Msamples/s, we show that the minimum-redundancy prefix coding is well suited for future satellite entropy coding standard.

Table of Contents

1. Introduction	1
2. Background Information	1
2.1 Minimum-Redundancy Prefix Coder	1
2.2 Simple Dual-Port RAM Operations	2
3. FPGA Hardware Realization	4
3.1 C0: Scheduling and Pipelining	4
3.2 C1: Overlapping RAM Operations	4
3.3 C2: Solving Complex Loop Controls	6
3.3.1 Predicate Read and Conditional Write	6
3.3.2 What Seems Difficult May Turn Out Easy	8
3.4 C3: Generating Codeword and Overlapping Operations across Stages	8
4. Further Optimization Techniques	9
4.1 Finding More Overlapping Operations across Sub-Stages	9
4.2 Resource Bound and Resource Duplication	10
5. Experiment	11
5.1 Experimental Setup	12
5.2 Performance Metrics	12
5.3 Resource Usage and Power Consumption	13
5.4 Design Verification	14
6. Conclusion	14
7. References	14

1. Introduction

Imagery data captured by satellite sensors provides valuable information for weather forecast, geo-environmental monitoring, and oil drilling. However, efficient compression of satellite images requires a data coding scheme with the characteristics of low-complexity, high compression ratio, and provides lossless image quality. In search for an efficient data coding scheme for satellite images therefore becomes important and may significantly influence the efficiency and quality of the compressed data.

In 1971, Rice [2] proposed a low-complexity, prefix-free entropy coder. As opposed to Huffman coding [3], decoder of Rice coding scheme does not require original codeword table. Although Rice coding is recommended by the Consultative Committee for Space Data Systems (CCSDS), it is not optimal in practice. Recently, a new minimum-redundancy prefix coding scheme is proposed in [1], which provides a linear-time, prefix-free coding for space data compression. Similar to the Rice coding, no codeword table is required to send to the decoder in minimum-redundancy prefix coding scheme.

To show the effectiveness and hardware-friendliness of the minimum-redundancy coding algorithm, in this paper, a high throughput, low power FPGA realization of the minimum-redundancy prefix coder will be discussed. Optimization methods such as loop transform, pipelining, and scheduling techniques will be applied to improve the performance/throughput of the hardware design. The rest of the paper is organized as follows: Section 2 describes the minimum-redundancy prefix coder and Altera FPGA RAM block operations. Sections 3 and 4 propose the FPGA hardware realization and optimization techniques. Section 5 comments on the experimental results, and finally, Section 6 concludes the paper.

2. Background Information

In this section, we will give a brief introduction to the minimum-redundancy prefix coder. In addition, we will discuss the functional characteristics of a simple dual-port RAM, which can be found in most major FPGA vendor and is essential in the realization of the minimum-redundancy prefix coder.

2.1 Minimum-Redundancy Prefix Coder

The minimum-redundancy prefix coder proposed in [1] provides a linear-time, prefix-free entropy coding targeted for real-time space data compression. It has the characteristics of low complexity, high

compression ratio, and most importantly, it is lossless during data coding. The coding algorithm is comprised of five stages, C0 to C4. The first stage C0, the preprocessing stage, performs the differential pulse code modulation (DPCM) and positive mapping. This step is optional and can be replaced by other preprocessing schemes to improve data coding and compression ratio. The C1 stage, the linear-time frequency table builder, generates the statistics of the preprocessed data twice and sorts the occurrence frequency of each symbol in linear time. In the C2 stage, code length for each symbol is determined and its corresponding codeword is generated in C3. The C4 stage maps the symbol to codeword, and performs the bit compacting before sending out the compressed result. Fig. 1 shows the data processing flow of the minimum-redundancy prefix coder.

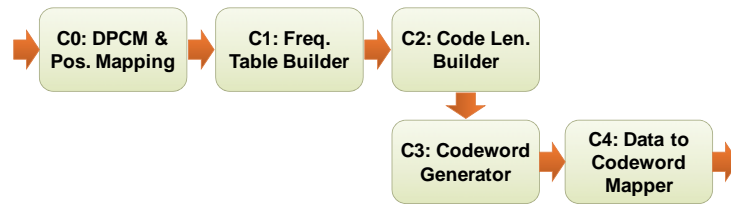


Fig. 1. Data processing flow diagram for the minimum-redundancy prefix coder [1].

2.2 Simple Dual-Port RAM Operations

To increase the level of parallelism and open up pipelining capability, in this design we use simple dual-port RAMs on all of the memory blocks. A simple dual-port RAM is comprised of a read-address port, a write-address port, a data input port, a data output port, and write enable and clock input. A representative view of a simple dual-port RAM is shown in Fig. 2(a).

To make the most out of the simple dual-port RAM, it is very important to fully understand its read and write operations. Fig. 2(b) and (c) show the read and write behaviors of a simple dual-port RAM. Two observations must be kept in mind. First, during a read operation, the memory returns the requested data one cycle after the address is sent in. Second, if we read and write the same address in the same cycle, the returned data will be the old data stored in that address. It is not until the second read of that address would return the newly written data. Although some FPGA vendors such as Altera [4] provide special hardware description language syntax to support the write-through behavior (by automatically adding the bypass logic), to make the design portable to all FPGA manufacturers and future ASIC flow, we figure out

possible data dependency issues and design the corresponding feed forwarding logic. Fig. 3 shows an example of overlapping operations and possible data forwarding using a simple dual-port RAM, where *A* refers to sending of address stage, *D/M* refers data received and modification stage, and *W* refers to the memory write stage. We will elaborate more on this in the later sections.

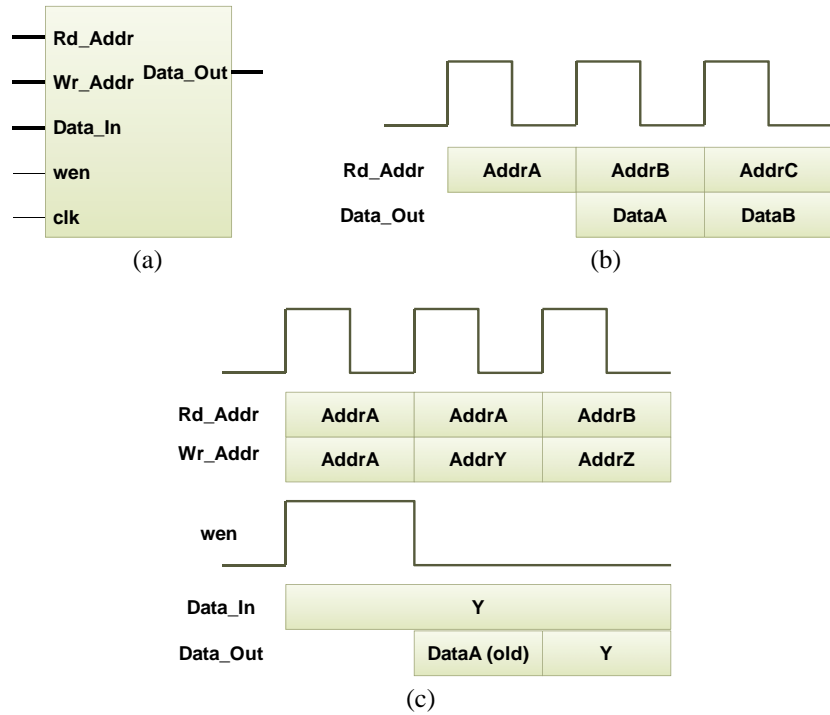


Fig. 2. Illustration of simple dual-port RAM operations. (a) Representative symbol of a simple dual-port RAM. (b) Read operation. (c) Write operation.

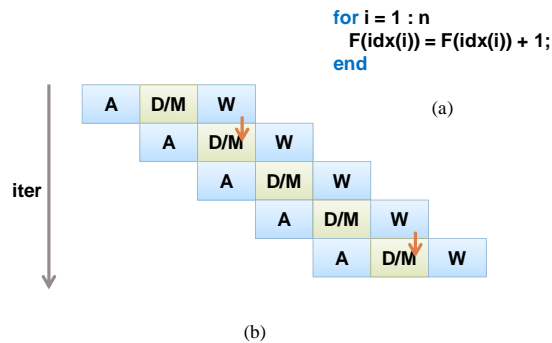


Fig. 3. Example of overlapping read and write operations by using a simple dual-port RAM. (a) Example loop updating sequence. (b) Example of overlapping and data forwarding.

3. FPGA Hardware Realization

In this section, the first prototype hardware design for the minimum-redundancy prefix coder is proposed. Special features and highlights for each individual stage will be presented in each subsection. Performance evaluation and further optimization techniques will be presented in Section 4 and 5.

3.1 C0: Scheduling and Pipelining

In our experimental setup, two 16-bit samples are read into the preprocessing stage each cycle. The preprocessing stage C0 then performs differential pulse code modulation (DPCM) and positive mapping on these two samples. To avoid unnecessary fetching of redundant data samples, we use a register to temporarily store the sample at higher address. Therefore in any iteration, the preprocessing stage is able to consume two data samples and generate two corresponding DPCM and positive mapped results. These two results are later stored in two different memory modules to improve throughput for later stages. Fig. 4 shows the data flow of this behavior. Note that extra pipeline stage is inserted in the DPCM and positive mapping to shorten the cycle time. Also note that the sample at address 0 corresponds to the DC value and will be stored as is in memory.

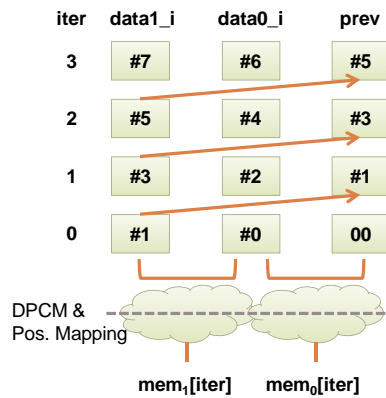


Fig. 4. Illustration of dataflow and pipeline stage for C0.

3.2 C1: Overlapping RAM Operations

The linear-time frequency table builder stage contains five software loops. The first loop generates the statistics of each symbol on the preprocessed data, and stored in symbol-frequency table F . The second loop performs similar operation, but now uses the frequency value in F as input value and generates the symbol-frequency table F_2 . The third and fourth loop performs data arrangements on F_2 and generates the

index table Idx . Lastly, the fifth loop generates a non-increasing, frequency-sorted table F_s based on table Idx and both table are sent out for reference at later stages. Since the first four loops depend on the completion of its previous loop, only intra-loop, inter-iteration parallelism can be explored.

Fig. 5 shows an example scheduling of pipelined memory operations for pursuing inter-iteration parallelism on a simple dual-port RAM device. The software example resembles a simple frequency counter for incoming symbols, which is performed twice in the frequency table builder stage. We can see that even in this simple example, for each incoming symbol, we need to take care of two possible data feed forwarding paths in order to update memory with the correct values. Under this pipelined scheme, we can see that although each iteration still spans three cycles, with n input data, we are able to finish the whole loop in $n+2$ cycles, which is a great saving from the $3n$ cycles. Fig. 6 shows a detailed schedule of applying the similar technique on the fourth loop in C1.

Apart from the memory techniques, extra parameters such as number of non-zero symbols, maximum symbol for F and F_2 are recorded to avoid unnecessary cycle waste.

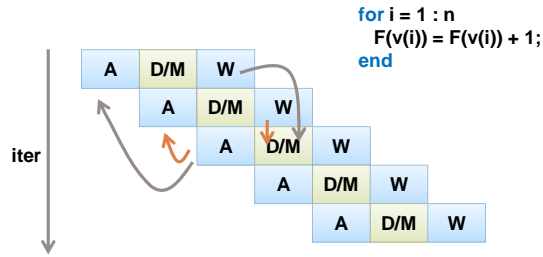


Fig. 5. Example of pipelining the memory operations and feed forwarding.

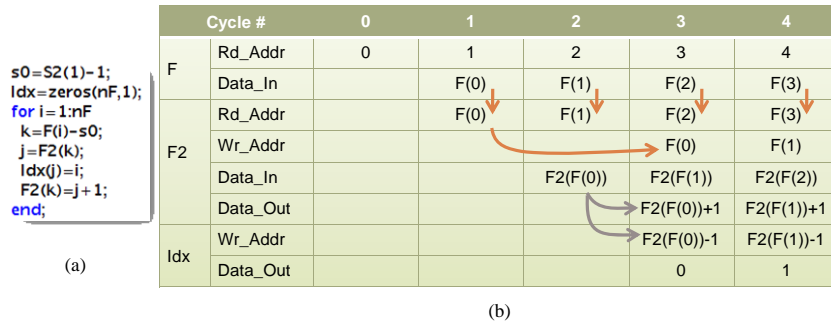


Fig. 6. C1d schedule. (a) Matlab implementation, and (b) its corresponding schedule.

3.3 C2: Solving Complex Loop Controls

The code length builder stage, or C2, contains three loops. These three loops build up relationship and set depth information for each symbol, and finally determine the code length for each symbol. Similar to C1, each loop in C2 depends on the information set by their corresponding previous loop, which means that only intra-loop parallelism can be exploited. In addition, the complex conditional behavior of each iteration further complicates the hardware circuit design. Fig. 7 shows the Matlab version of the most challenging part in the minimum-redundancy prefix coder. Detailed analysis and hardware design will be presented later in this section.

```
% setting parent pointers for all r
l=zeros(n1,1); l(n1)=F(n1)+F(n);
i=n1; f=n2;
for k=n2:-1:1,
    if f < 1 || l(i) < F(f)
        T=l(i); l(i)=k; i=i-1;
    else
        T=F(f); F(f)=k; f=f-1;
    end;
    if f < 1 || (i > k & l(i) < F(f))
        l(k)=T+l(i); l(i)=k; i=i-1;
    else
        l(k)=T+F(f); F(f)=k; f=f-1;
    end;
end;
```

(a)

```
% setting depths for (n-1) internal nodes
l(1)=0;
for k=2:n1,
    l(k)=l(l(k))+1;
end;
```

(b)

Fig. 7. Matlab version of C2 to demonstrate complicated loop control. (a) C2a: first loop, and (b) C2b: second loop.

3.3.1 Predicate Read and Conditional Write

To tackle the complex loop control problem as demonstrated in Fig. 7(a), we propose a predicate read, conditional write scheme to fuse the two if-else conditions. Fig. 8 shows a 3 cycle/iteration pipelined memory operation for the I table and F table. This is the tightest possible pipelining scheme. Note that in each iteration, the W cycle with parenthesis refers to the conditional write operation, which means that the write operation is initiated according to the result of the two *if* condition. Furthermore, two predicate read operations are performed each iteration, and same data may be re-read again in the following iterations. A detailed I table schedule for this 3 cycle/iteration scheme is demonstrated in Fig. 9.

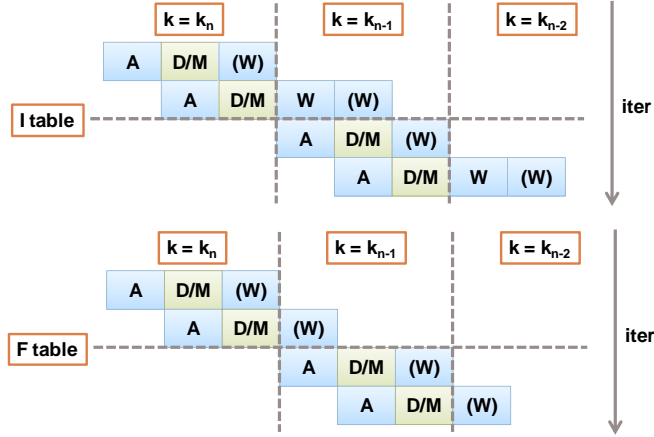


Fig. 8. A 3 cycle/iteration design for C2a.

State	0	1	2	0	1	
Rd_Addr	c_i	c_{i-1}		c_{i-2}		condA _i condB _i
Wr_Addr			c_i	k_1	c_{i-1}	
Data_In		$l(c_i)$	$l(c_{i-1})$			
Data_Out			k_1	$T+l(c_{i-1})$	k_1	
Rd_Addr	c_i	c_{i-1}		c_{i-1}		condA _i condB _f
Wr_Addr			c_i	k_1	-	
Data_In		$l(c_i)$	$l(c_{i-1})$			
Data_Out			k_1	$T+F(c_i)$	-	
Rd_Addr	c_i	c_{i-1}		c_{i-1}		condA _f condB _i
Wr_Addr			-	k_1	c_i	
Data_In		$l(c_i)$	$l(c_{i-1})$			
Data_Out			-	$T+l(c_i)$	k_1	
Rd_Addr	c_i	c_{i-1}		c_i		condA _f condB _f
Wr_Addr			-	k_1	-	
Data_In		$l(c_i)$	$l(c_{i-1})$			
Data_Out			-	$T+F(c_{i-1})$	-	

Fig. 9. C2a *I* table schedule (3 cycle/iteration version).

The 3 cycle/iteration scheme provides the tightest possible schedule and the fewest cycle count to complete execution of the loop in Fig. 7(a). The down side is that it generates long critical path which decreases the maximum working frequency by 20%. In order to solve this problem and to provide a simple clock scheme (single clock) for our prefix coder design, we also propose a 4 cycle/iteration scheme. Detailed pipelined memory operation for the 4 cycle/iteration scheme is shown in Fig. 10. Note that the detailed *I* and *F* table schedule can be done similarly as the 3 cycle/iteration example and thus omitted here. Tradeoff between the two schemes will be discussed in Section 5.

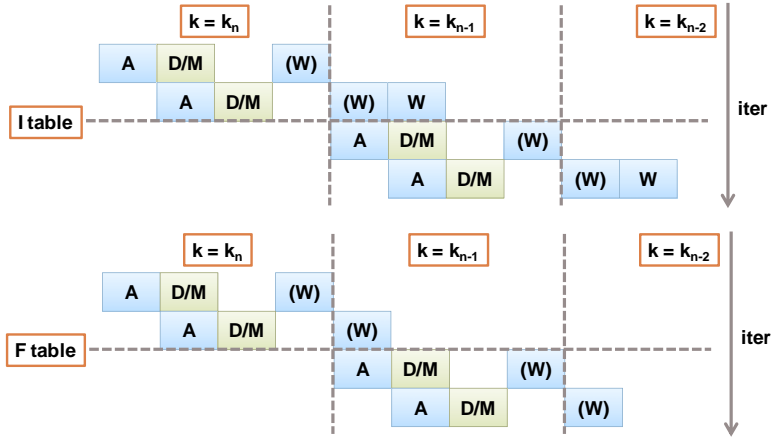


Fig. 10. A 4 cycle/iteration design for C2a.

3.3.2 What Seems Difficult May Turn Out Easy

The second loop in C2 may also contain nasty inter-iteration dependency problem, if the data within the memory is not sorted. The same table is read twice and then a write is initiated. Fortunately enough, the *I* table generated in C2a contains only non-increasing data. This results in the detailed schedule as shown in Fig. 11. Note that this is the tightest possible schedule, and only the dashed line part need to take care of possible data bypassing problem.

Cycle #	2		3		4	
State	0	1	0	1	0	1
Rd_Addr	2	I(2)	3	I(3)	4	I(4)
Wr_Addr		1		2		3
Data_In		I(2)	I(I(2))	I(3)	I(I(3))	I(4)
Data_Out		0		I(I(2))+1		I(I(3))+1

Fig. 11. C2b *I* table schedule.

3.4 C3: Generating Codeword and Overlapping Operations across Stages

The codeword generator, or C3, takes in the codeword length table determined in C2 and generate the corresponding codeword. The codeword generation logic is shown in Fig. 12. When current codeword length is equal to the previous codeword length, previous codeword plus one is assigned as the codeword for current symbol. Otherwise we assign previous codeword left shift by one plus two as the codeword for current symbol. Note that we arrange the code length information and codeword in one codeword table

entry to facilitate determination of codeword in C4. Also we rearrange the codeword to its corresponding symbol address to allow single table reference in C4 and save the worst case 256 cycles.

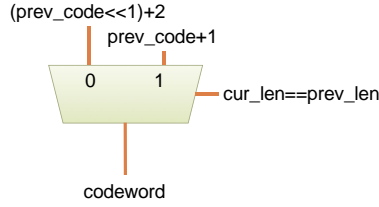


Fig. 12. Codeword generator logic.

4. Further Optimization Techniques

Based on the first prototype hardware design described in Section 3, we further optimize the design by resolving the resource bound problem and by overlapping more operations across stages.

4.1 Finding More Overlapping Operations across Sub-Stages

Due to the difference in nature in writing a software program and a hardware program, simultaneous or parallel execution of two different sub-stages may be possible and should be pursued in designing hardware. In 3.4 we talked about one possibility, and here is another example. Recall in 3.2 we proposed a detailed schedule of C1d. By carefully examining the original software implementation, it is not hard to find out that all necessary information for the fifth loop of C1 is available and can be written to the sorted frequency table F_s simultaneously. This saves another worst case cycle count by 256 and a small amount of performance gain is obtained. Fig. 13 shows the combined C1d and C1e scheduling.

Cycle #		0	1	2	3	4
F	Rd_Addr	0	1	2	3	4
	Data_In		F(0)	F(1)	F(2)	F(3)
F2	Rd_Addr		F(0)	F(1)	F(2)	F(3)
	Wr_Addr				F(0)	F(1)
F2	Data_In			F2(F(0))	F2(F(1))	F2(F(2))
	Data_Out				F2(F(0))+1	F2(F(1))+1
Idx	Wr_Addr				F2(F(0))-1	F2(F(1))-1
	Data_Out				0	1
Fs	Wr_Addr				F2(F(0))-1	F2(F(1))-1
	Data_Out				F(0)	F(1)

Fig. 13. Overlapped C1d and C1e scheduling for performance improvement.

4.2 Resource Bound and Resource Duplication

So now we have the first hardware prototype design. Is there a way to improve the performance to a higher level? Let us take a peek at the performance metric shown in Fig. 16. It is soon found out that the most time consuming part in this prototype design is the first loop in C1, or C1a. Is there a way to reduce the required cycle count to half, such that the overall performance would improve significantly?

The focus now turns to how to obtain speedup on the C1a. We obtain 2 DPCM and positive mapped symbols from C0 while we are currently using 2 cycles to handle them. Are we able to use them both in a single cycle?

Refer once again to the pipelined memory operation as shown in Fig. 5. We can see under this pipelined scheduling, the resource utilization (the read and write address port) is equal to 1 and cannot squeeze out more performance. One question arises: Are we using the correct memory? Can we obtain better performance if we replace it with a true dual-port RAM?

Consider again the simple frequency counter example. Fig. 14(a) shows the desired memory operation scheduling to handle the two symbols in each cycle. But this is not achievable with a true dual-port RAM because it cannot sustain two read and writes in one cycle. Resource conflict arises. The tightest possible scheduling is shown in Fig. 14(b). In this case, we are able to handle n symbols in n cycles. On the other hand, the tightest possible schedule for a simple dual-port RAM is shown in Fig. 14(c), which n symbols are completed in $n+2$ cycles. Since n is large (2048 in our case), the saving of 2 cycles is not significantly enough for us to switch from the simple dual-port RAM to true dual-port RAM.

So now we need to rethink about the meaning behind the simple frequency counter. A simple frequency counter counts the number of times of the occurrence of a symbol. Since we obtain two symbols from C0 each cycle, why not count them separately, and add the result up for later stages? This turns out to be the solution to our problem. The new pipelined memory operation is shown in Fig. 15. Notice that under this scheme, we need to spend one extra memory and adder to generate the correct result. When the later stages refer to this memory, same read address is supplied to both memory modules and their outputs are added together as the total frequency of that symbol.

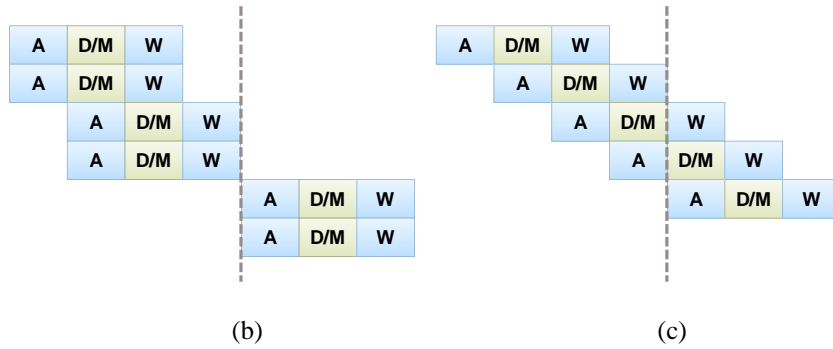
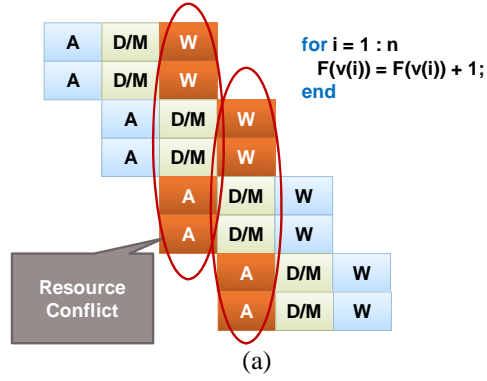


Fig. 14. Handle 2 symbols each cycle. (a) Desired pipelined memory operation. (b) Tightest possible schedule for a true dual-port memory. (c) Comparison to the tightest possible schedule for a simple dual-port memory.

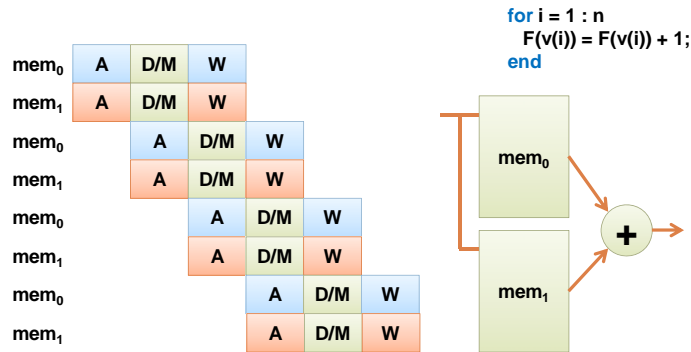


Fig. 15. Illustration of resource duplication to handle the simple frequency counter resource bound problem.

5. Experiment

In this section, environment setup for the minimum-redundancy prefix coder is presented. Based on the environment setup, we report the performance metrics, resource usage and power consumption of our proposed design.

5.1 Experimental Setup

The proposed implementation is entirely designed in Verilog hardware description language. We use the Altera Stratix III EP3SL150F1152C2 FPGA as the carrier for our design. Synthesis, timing analysis and power analysis are done in Altera Quartus II 9.0, and functional and post place and route timing simulation are performed in ModelSim SE 6.3f.

5.2 Performance Metrics

Fig. 16 shows the performance metrics for three different schemes: (1) 3 cycle/iteration C2a, (2) 4 cycle/iteration C2a, and (3) 4 cycle/iteration C2a + optimization methods discussed in Section 4. Note that we only present worst case performance in this report. As mentioned in Section 3, three special parameters are stored in order to improve the throughput, and all stages except for C0, C1a, C1b, and C4 require variable number of cycles to complete. Average throughput of the design should be investigated using the real-world satellite data in the near future. Throughputs and maximum working frequencies of all three schemes of the design are shown in Fig. 17. Note that when use only one instance of the proposed hardware, we are able to outperform the CCSDS throughput requirement by a maximum of 3.928x. Since the algorithm is line-by-line independent, we can insert arbitrary number of instances of the same hardware and perform the coding together, as long as they fit into the available FPGA resource.

Stage		3 c/i		4 c/i		4 c/i + chap 4		Description
		worst	lat	worst	lat	worst	lat	
C0	-	1024	3	1024	3	1024	3	input 2048 samples
C1	a	2048	4	2048	4	1024	4	
	b	256	2	256	2	256	2	
	c	1025*	3	1025*	3	1025*	3	< max(freq)
C2	d	256	4	256	4	256	4	max(non-zero symbol)
	e	256	4	256	4	-	-	# of non-zero symbols
	a	256*3	4+3	256*4	5+4	256*4	5+4	if(# of non-zero symbols) > 2 (# of non-zero symbols-2)*4
C3	b	256*2	2	256*2	2	256*2	2	if(# of non-zero symbols) > 2 (# of non-zero symbols-2)*2
	c	256	3	256	3	256	3	# of non-zero symbols
C4	-	256	2	256	2	256	2	# of non-zero symbols
C4	-	(2048)	(3)	(2048)	(3)	(2048)	(3)	ongoing
Cycle #		8742		9000		7715		

Fig. 16. Performance metrics for three different schemes.

Scheme	3 cycle/iteration	4 cycle/iteration	4 cycle/iteration + chap 4 opt.
Max. working freq. (MHz)	200	284.09	295.95
Throughput (Msamples/s)	46.854	64.646	78.562
Throughput (Gbits/s)	0.750	1.034	1.257
Speedup w.r.t. CCSDS recom.	2.343 x	3.232 x	3.928 x

Fig. 17. Throughput and maximum working frequency comparison of the three proposed schemes.

5.3 Resource Usage and Power Consumption

Resource usage and power consumption are both reported in the Altera Quartus II software. Resource usage information can be obtained after the place and route operation. Power consumption is reported using the PowerPlay Power Analyzer Tool provided in the Quartus II software. To obtain power consumption number with higher estimation confidence, a post place and route simulation waveform is supplied to the power analyzer tool.

Fig. 18 shows the resource usage and power consumption before and after applying the optimization mentioned in Section 4. We can see that with the optimization scheme, the number of registers decreases while the block memory bits increases. Register reduction is due to the fact that C1e is removed, and increased number in total memory bits is the duplication of memory in C1a. Note that although total power increases, the power/sample indicator decreases. The power/sample indicator stays well within the CCSDS recommendation on power consumption for new coding scheme (0.5W/Msamples/s). The result also suggests further power reduction is possible if further improvement in throughput can be achieved.

Scheme	4 cycle/iteration	4 cycle/iteration + chap 4 opt.
Logic utilization (Comb ALUTs)	1,972/113,600 (2%)	1,950/113,600 (2%)
Total registers	1692	1642
Total block memory bits	92,672/5,630,976 (2%)	97,280/5,630,976 (2%)
Total power estimation (mW)	808.99	823.92
Power estimation (W/Msamples/s)	0.0125	0.0105

Fig. 18. Resource usage and power consumption.

5.4 Design Verification

The correctness of the design is verified by Verilog testbench file in ModelSim SE 6.3f. Input and output are sent in and collected by the testbench file. The output is then verified with the expected result generated by the Matlab software. Both functional and post place and route timing simulation are performed to verify the correctness of the design. Fig. 19 shows an example verification scene.

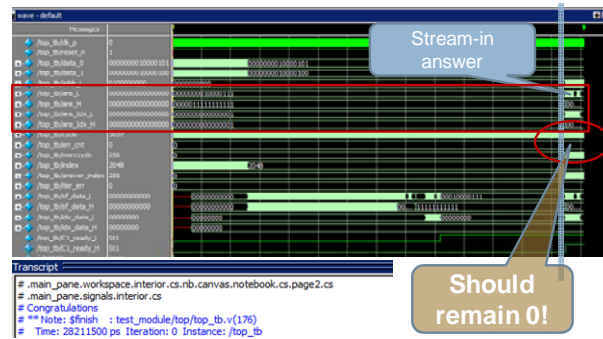


Fig. 19. ModelSim software verification scenario.

6. Conclusion

In this paper, we proposed a high performance FPGA hardware implementation of the minimum-redundancy prefix coder. Experimental setup and techniques used were discussed in detail. Performance comparison between different design trade-offs were evaluated. Results show that the proposed design is able to achieve a high throughput of 78.562 Msamples/s at the low power consumption of 0.0105 W/Msamples/s, which surpassed the CCSDS recommendation on throughput and power with the mainstream FPGA.

We are still working on the hardware implementation on C4, and whole circuit design verification with real-world satellite data. More accurate average throughput performance and power consumption can be expected in near future.

7. References

- [1] B. Huang, "Fast Minimum-redundancy Prefix Coding for Real-Time Space Data Compression," Proc. SPIE, 6683, 66830H, 2007.
- [2] Lossless Data Compression. Recommendation for Space Data System Standards, CCSDS 121.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, May 1997.

- [3] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proc. Inst. Radio Eng., vol. 40, iss. 9, pp. 1098-1101, Sept. 1952.
- [4] "Stratix III Device Handbook." [Online].
Available: http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf
- [5] "Quartus II Handbook Version 9.0." [Online].
Available: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf