Hardware Acceleration of the Robust Header Compression (RoHC) Algorithm

Mohammed Al-Obaidi, aso10mma@student.lu.se
Harshavardhan Kittur, aso10hki@student.lu.se

August 28, 2012

This Page is Intentionally Left Blank

Preface

This Masters thesis summarizes our research work carried out at Ericsson AB, Linkoping, in close collaboration with Product Development Unit LTE. The main contributions for this thesis are:

- Hardwaware-Software Codesign of RoHC by Harshavardhan Kittur.
- Full Hardware Design of RoHC by Mohammed Al-Obaidi.

Acknowledgements

This thesis was made available and financially supported by Ericsson AB in Linköping. We are grateful to Stefan Backlund, who gave us the opportunity to do this thesis, and provided us with all the needful to make our work in Ericsson AB smooth and pleasant.

The writing of this thesis was one of the most significant challenge in our academic life. Without the support, guidance, and patience of the following people it would not have been done at this quality. Our deepest gratitude to all the people who helped us throughout this thesis.

- Our two supervisors, Håkan Anderson in Ericsson AB-Linköping and Professor Viktor Öwall in Lund university, who helped us to hurdle all the obstacles during this thesis. We appreciate their time and patience in correcting the writing of this report. We are so happy for getting the chance to work with them in this project and we will never forget their sincere concern and patience.
- Bo Eriksson and Niclas Hallqwist in Ericsson AB-Linköping, who attended our weekly supervision meetings, provided us with their advices, and were always confident of our work.
- John Fong and Adrian Turcanu from Ericsson AB-Ottawa, who helped us getting familiar with the RoHC algorithm, and patiently answered our questions.
- Göran Westerlund and Adam Ashby from Altera, who provided us with the FPGA board.
- Orges Balla and Johan Karlsson, whose feedbacks on the report were highly appreciated.
- Last but not the least, many thanks to our families and friends for their support, inspiration, and love.

Abstract

With the proliferation of Long Term Evolution (LTE) networks, many cellular carriers are embracing the emerging field of mobile Voice over Internet Protocol (VoIP). The robust header compression (RoHC) framework was introduced as a part of the LTE Layer 2 stack to compress the large headers of the VoIP packets before transmitted over LTE IP-based architectures. The headers, which are encapsulated Real-time Transport Protocol (RTP)/User Datagram Protocol (UDP)/Internet Protocol (IP) stack, are large compared to the small payload. This header-compression scheme is especially useful for efficient utilization of the radio bandwidth and network resources.

In an LTE base-station implementation, RoHC is a processing-intensive algorithm that may be the bottleneck of the system, and thus, may be the limiting factor when it comes to number of users served. In this thesis, a hardware-software and a full-hardware solution are proposed, targeting LTE base-stations to accelerate this computationally intensive algorithm and enhance the throughput and the capacity of the system. The results of both solutions are discussed and compared with respect to design metrics like throughput, capacity, power consumption, chip area and flexibility. This comparison is instrumental in taking architectural level trade-off decisions in-order to meet the present day requirements and also be ready to support future evolution.

In terms of throughput, a gain of 20% (6250 packets/sec can be processed at a frequency of 150 MHz) is achieved in the HW-SW solution compared to the SW-Only solution by implementing the Cyclic Redundancy Check (CRC) and the Least Significant Bit (LSB) encoding blocks as hardware accelerators . Whereas, a Full-HW implementation leads to a throughput of 45 times (244000 packets/sec can be processed at a frequency of 100 MHz) the throughput of the SW-Only solution. However, the full-HW solution consumes more Lookup Tables (LUTs) when it is synthesized on an Field-Programmable Gate Array (FPGA) platform compared to the HW-SW solution. In Arria II GX, the HW-SW and the full-HW solutions use 2578 and 7477 LUTs and consume 1.5 and 0.9 Watts, respectively. Finally, both solutions are synthesized and verified on Altera's Arria II GX FPGA.

Contents

List of Figures

List of Tables

List of Abbreviations

3GPP	3rd Generation Partnership Project
ALMs	Adaptive Logic Modules
ALUT	Altera Lookup Table
BER	Bit Error Rates
CISC	Complex Instruction Set Computer
CRC	Cyclic Redundancy Check
CSRC	Contributing Source Identifier
FIFO	First-In First-Out
FO	First Order
FPGA	Field-Programmable Gate Array
FS	Field Selection
HAL	Hardware-Abstraction Layer
IETF	Internet Engineering Task Force
IP	Intellectual Property
IP	Internet Protocol
IP-ID	Internet Protocol-Identification
IR	Initialization and Refresh
LOD	Leading-One Detector
LSB	Least Significant Bit
LSZD	Least Significant Zero Detector
LTE	Long Term Evolution
LUT	Lookup Table

MM	Avalon Memory Maped
NBO	Network Byte Order
O-mode	Bidirectional Optimistic Mode
PDCP	Packet Data Convergence Protocol
R-mode	Bidirectional Reliable Mode
RoHC	Robust Header Compression
RR	Receiver Report
RTCP	RTP Control Protocol
RTP	Real-time Transport Protocol
RTP-SN	Real-time Transport Protocol-Sequence Number
RTP-TS	Real-time Transport Protocol-Timestamp
SBT	Software Build Tools
SDVL	Self-Describing Variable Length
SN	Sequence Number
SO	Second Order
SR	Sender Report
SSRC	Synchronization Source Identifier
ST	Avalon streaming
SW	Sliding-Window
TS	Timestamp
U-mode	Unidirectional Mode
UDP	User Datagram Protocol
UID	User ID
VoIP	Voice over Internet Protocol
vref	Reference value
WLSB	Window-based LSB

This Page is Intentionally Left Blank

Chapter 1

Introduction

With many cellular carriers following the evolution of telephony over the internet, they are now seriously considering transmission of voice over all IP based LTE networks via VoIP. It is a known problem that VoIP packets transmitted via the RTP/UDP/IP stack, have relatively small payloads compared to the overhead of the packet headers enclosed by the respective protocol stack. The header overhead is about 40-60 bytes (depending on IPv4 or IPv6 respectively) while the minimum size payload is 20 bytes, which creates 200-300% overhead. This overhead is expensive, especially on radio links that are affected by high Bit Error Rates (BER). Here, efficient utilization of the sparse radio bandwidth for useful payload data is of utmost importance. Redundant packet-header data in the stream is a waste of resources. This creates a need for some kind of header-compression technique for efficient utilization of the bandwidth and network resources to deliver a unit of payload data.

The RoHC framework was introduced by the Internet Engineering Task Force (IETF) in [1] and adopted by the 3rd Generation Partnership Project (3GPP) as a solution to this problem. This header-compression method is based on the principle that there is a significant redundancy between the headers of packets belonging to the same stream. By sending the static information of the header fields initially and then continuously sending only the dynamic parts of the headers, the size of the headers can be compressed significantly. The major advantage of RoHC over other header-compression techniques is its robustness. It can tolerate packet loss and residual errors on the link without losing additional packets or introducing additional errors.

Traditional base-stations have implementations in software, with some parts accelerated in hardware. In [2], a profiling of the most complex algorithms in LTE layer 2 was done using an ARM-based mobile hardware platform and with LTE data rates of about 100 Mbps. It was shown that the traditional hardware-software partitioning is not a tractable solution anymore and a new, sophisticated partitioning is needed. It was shown that the Packet Data Convergence Protocol (PDCP) sub-layer was the most computationally intensive in LTE Layer 2. Moreover, it was shown that the RoHC algorithm is taking most of the processing power (71%) in this sub-layer, which makes it a suitable target for hardware acceleration. In [3], an analysis of the memory bandwidth requirements was done and the hardware implementation of some RoHC functions was proposed. The authors showed that, in the next generation of mobile networks, network processors need to be augmented with more hardware in order to support the processing complexity of the RoHC algorithm.

Also, for base stations to support high capacity (number of users), there is a hard deadline on the computation of different algorithms. RoHC can be considered one of the performance-critical algorithms in the LTE stack. In this thesis, a Hardware-Software co-design and a Full Hardware-based solution are proposed to determine the improvement in the performance with respect to the degree of the HW-SW partitioning. Both solutions are targeting the RoHC algorithm in the LTE base stations.

The rest of this report is organized into eight chapters. In chapter 2, RoHC is discussed with respect to the LTE protocol stack and in chapter 3, the RoHC framework is briefly reviewed. In chapter 4, the design of hardware accelerators that are used in both solutions are discussed. In chapter 5, the HW-SW co-design of RoHC and the corresponding results are discussed. In chapter 6, the full hardware implementation of this algorithm is presented. Subsequently, results from both solutions are compared and presented in chapter 7. Finally, the conclusions are drawn in chapter 8.

Chapter 2

Background of RoHC in LTE System

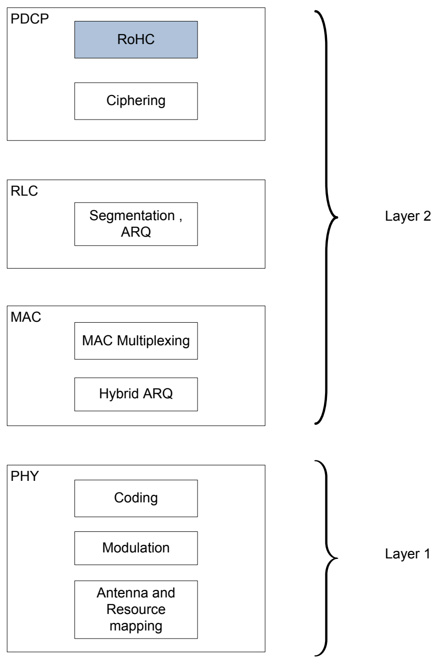


Figure 2.1: RoHC in the LTE protocol architecture

LTE is the fourth generation of high-speed and high-capacity wireless communication standards for mobile and data terminals specified by the release of the 3GPP standard [4]. The protocol architecture is bifurcated into control-plane which deals with signaling and control functions, and data-plane, which deals with the actual user data transmission. RoHC is the standard for header compression as proposed by IETF [1]. RoHC resides

Table 2.1: RoHC compression gain

Protocol stack	Uncompressed header size (bytes)	Min. Compressed header size (bytes)	Compression gain(%)
IPv4/TCP	40	4	90
IPv4/UDP	28	1	96
IPv4/UDP/RTP	**40**	**1**	**97.5**
IPv6/TCP	60	4	93
IPv6/UDP	48	3	93
IPv6/UDP/RTP	**60**	**3**	**95**

in the PDCP sub-layer 2 in the data-plane of the LTE protocol architecture, as shown in Figure 2.1 [5].

The major use of the RoHC framework is to compress the relatively large packet header, which would be a big overhead especially for small packets like VoIP packets, in-order to ensure efficient use of the radio interface and the available bandwidth. LTE is an all IP-based system where data are transmitted over a packet-switched networks to support a wide range of services in mobile handsets, VOIP is one of these services. IP-based traffic on radio links suffers high packet losses because of high bit error rates. Since the RoHC algorithm is designed to robustly handle the bad channel conditions, it is specifically useful in the wireless IP-based networks.

Real-time traffic like voice in VoIP packets, which are encapsulated in the IP/UDP/RTP protocol stack, has header sizes of 40 bytes (20 bytes for IPv4, 8 bytes for UDP, 12 bytes for RTP) to 60 bytes (40 bytes for IPv6, 8 bytes for UDP, 12 bytes for RTP) and the payload size is about 20 bytes (G.729 @ 8 Kbps) to 160 bytes (G.711 @ 64 Kbps) [1] depending on the type of encoding used. Thus, the header creates a large overhead but by using RoHC, the 40-60 byte header can be reduced down-to 1-3 bytes and thereby considerably reducing the overhead. The typical compression ratios for different protocol header is as shown in Table 2.1. We see that RoHC is most beneficial to compress VoIP packets encapsulated in IP/UDP/RTP headers.

RoHC is a new emerging field that has been studied recently after it was adopted by the LTE system. In [2], the LTE layer 2 protocols was processed on an ARM based mobile architecture to identify the most time critical algorithms. A single ARM1176 core processor was used to implement most of the layer 2 protocols in software whereas only the ciphering algorithm is accelerated in hardware. This hardware-software partitioning is a traditional implementation in current mobile handset architectures. It was shown by [2] that the layer 2 performance was limited by the computation power of the single-core processor and this traditional partitioning need to be changed to cope with the LTE requirements of layer 2 in next generation mobile devices. Moreover, It was shown that a major part of the computation power (71%) in LTE L2 protocol stack is consumed by the PDCP sub-layer. In the PDCP sub-layer, RoHC is identified as the major consumer of the computation power (71%) when the encryption in the PDCP

[1] `http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml`

sub-layer is disabled.

In [6], two implementations of RoHC, hardware and software, are presented by the same authors as in [2]. The effects of different cache sizes on the execution time of both solutions were studied. It was shown in [6] that the maximum performance and power saving were achieved with an instruction and data cache size of 16 kB in the embedded processor. Whereas, an improvement of 80% was achieved when the dedicated hardware accelerator is used with a relatively small cache compared to the software solution. The best execution time achieved by the dedicated hardware is around 40 μs whereas the software solution can process one packet in $80\mu s$.

In [3], a hardware acceleration of the main encoding method used in RoHC, LSB Encoding, is proposed. An analysis of the memory bandwidth requirement and its implication on the performance of the next generation network processors is studied. It was shown that a memory size of almost 264 bytes are required in the compressor side for each user. The analysis in [3] showed that in order to support a link speed of 2.5 Gbps, a memory bandwidth of 42 Gbps is needed. This bring the attention to the memory-bandwidth as one of the limiting factors in supporting high speed links.

Although many authors, such as in [2] and [3], have concluded that RoHC is needed to be accelerated in hardware, the industrial market for RoHC is still in pure software. In [7], a software implementation of RoHC is presented based on Intel microprocessor. Even with the use of a powerful Complex Instruction Set computer(CISC) such as Pentium 4, the number of flows that can be processed in [7] with a 100% processor load did not exceed the $13k$ packet flows. Thus we can expect that the industrial market of RoHC will follow the scientific recommendations soon, at least from the base station side which has to cater to a full capacity and support a large number of users.

Chapter 3

Summary of RoHC Framework Specifications

3.1 Classification of Header Fields

The basic principle of RoHC is that only a few header fields change randomly across a packet stream, while most of the fields change in a predictable way or not change at all. This makes header compression plausible. The header fields of a typical VoIP stack of IP/UDP/RTP are as shown in Figure 3.1. They can be classified into four different categories:

1. **Inferred**: These fields are never sent within a RoHC packet and can be inferred by the decompressor from the other fields or with the support of lower layers in the network stack.
2. **Static**: These fields are seldom sent within a RoHC packet since they never change during the lifetime of a packet stream.
3. **Static known**: These fields are never sent since they are constant in any packet stream.
4. **Dynamic**: These fields are of prime importance to RoHC to compress efficiently. They either change within a pattern or randomly.

The dynamic fields, which are of fundamental importance, have to be communicated either directly or derived from other fields for every packet. For RoHC based on the VoIP framework, the RTP Sequence Number (RTP-SN) is used to establish functions to other dynamic fields such as IP Identification (IP-ID), RTP Timestamp (RTP-TS). Hence, only RTP-SN needs to be reliably communicated while the other fields are derived from the RTP-SN functions. Other dynamic fields are changing in general but are expected to be constant during the lifetime of the packet stream. The static fields are

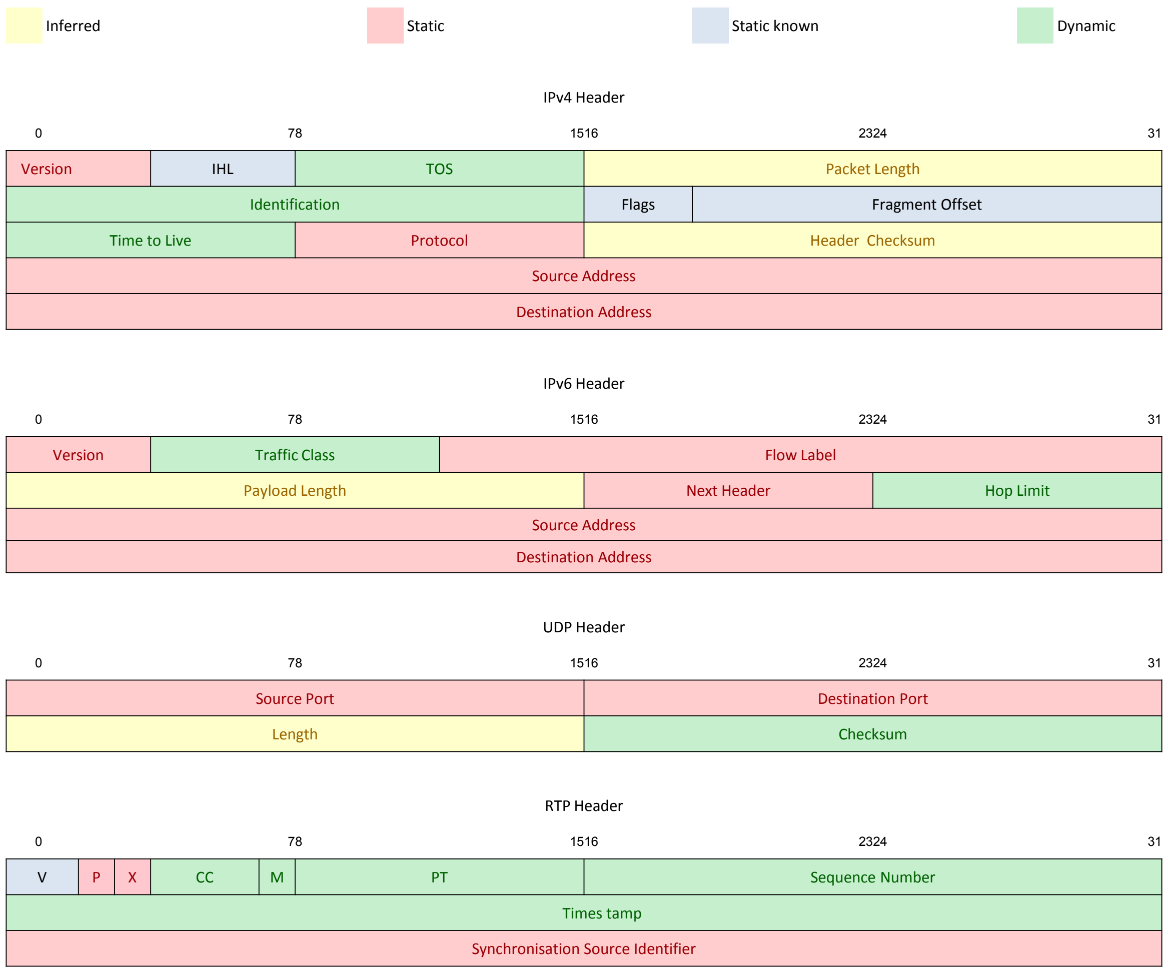


Figure 3.1: VoIP packet header fields

communicated using the uncompressed RoHC initialisation packets. Hence, all the fields of the IP/UDP/RTP headers are communicated either implicitly or explicitly during the packet flow.

3.2 Profiles

The RoHC profiles specify the schemes of both compression and decompression for a packet stream. In the RFC3095 specification [1], four different profiles are defined for different protocol stacks. These profiles are given below:

- **Profile 0**: For sending uncompressed packets. (This is the default scheme when the packets fail to fall within predefined RoHC profiles).

- **Profile 1**: This is for RTP/UDP/RTP compression.
- **Profile 2**: This is for UDP/IP compression.
- **Profile 3**: This is for ESP/IP compression.

3.3 Compressor and Decompressor

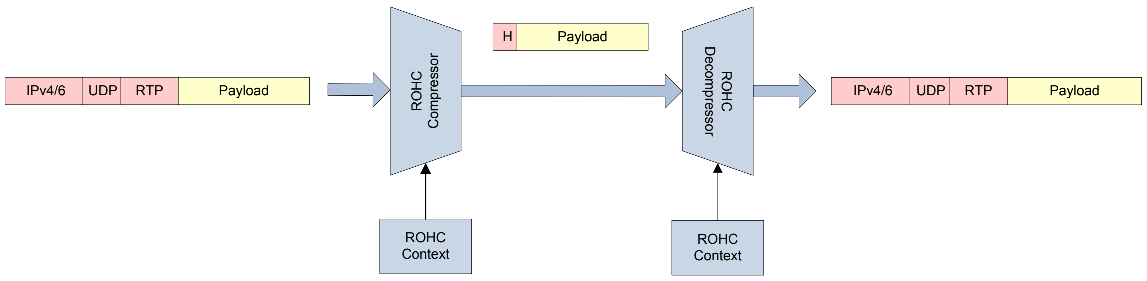


Figure 3.2: Compressor and decompressor blocks in a RoHC system

The major components of the RoHC framework are the compressor and decompressor as shown in Figure 3.2, which are present in PDCP Layer 2 of an LTE system. The compressor processes the incoming IP/UDP/RTP headers of the VoIP packet stream and compresses them. The compressed header is transmitted with the payload along the wireless link while the decompressor does the opposite for its decompression. Both the compressor and the decompressor use recently stored information from previous packets for a particular stream for the compression and the decompression processes. The stored information in the compressor or decompressor is referred to as the context of that compressor or decompressor, respectively. The contexts of the compressor and the decompressor must be consistent, to ensure a successful uninterrupted sequence of compression and decompression.

3.4 Modes

There are three modes of operation as shown in Figure 3.3 which determine the logic of state transitions and state actions. Depending on the existence of a feedback channel and the amount of feedback sent by the decompressor to the compressor, three modes are defined in the RFC3095 specification [1]. These modes: Unidirectional mode, Bidirectional optimistic mode, and Bidirectional reliable mode are briefly explained below. Transition between modes is possible and occurs when the decompressor triggers the change with the help of feedback packets.

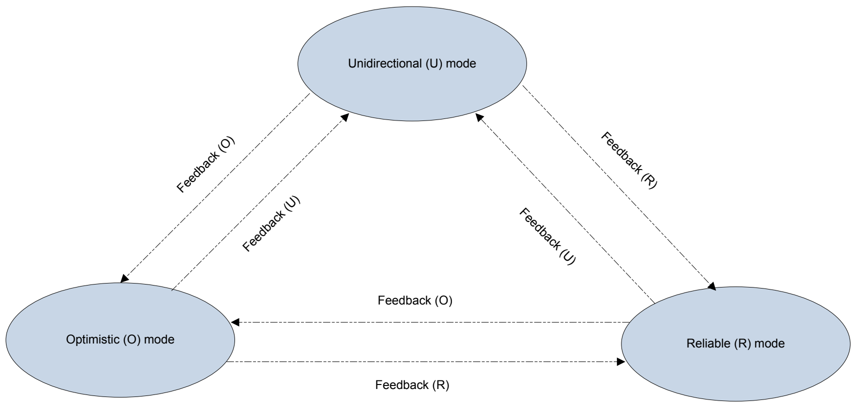


Figure 3.3: Modes of operation in a RoHC system

3.4.1 Unidirectional Mode (U-mode)

- This mode is for simplex channels which have no feedback capability; packets are sent in one direction from the compressor to the decompressor.
- In this mode, state transitions occur because of periodic timeouts and retransmissions because of irregularities in the header fields in the compressed packet streams from the compressor to the decompressor.
- The compressor in RoHC must start in this mode as the feedback capability of the channel is not known.
- In case of a packet loss or an unrecoverable error in the received compressed packet, the context of the decompressor might get invalidated and the sequence of decompression fails. In this mode, the decompressor waits for an uncompressed packet to resynchronize the context. The uncompressed packets are sent periodically and if the compressor detects a change in one of the static fields in the packet to be compressed.

3.4.2 Bidirectional Optimistic Mode (O-mode)

- This mode is similar to the unidirectional mode, but instead of the periodic timeout and the retransmission of uncompressed packets the feedback channel is used sparingly by the decompressor to send error recovery requests and acknowledgements of significant context updates.

3.4.3 Bidirectional Reliable Mode (R-mode)

- In this mode the feedback channel is used more often to acknowledge all the context updates. This mode employs stricter logic to maintain context synchronization between the compressor and decompressor.
- This mode maximizes robustness against loss propagation and damage propagation.

3.5 Compressor States

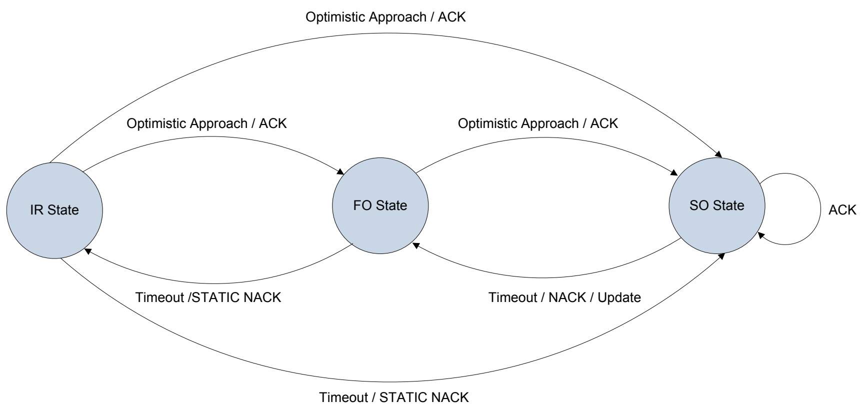


Figure 3.4: Compressor state machine

The compressor has three states as shown in Figure 3.4 - IR (Initialization and Refresh), FO (First order) and SO (second order) states. The compressor starts from the lowest compression state (IR) and gradually transits to higher-order states.

A forward state transition occurs when the compressor is confident that the decompressor has correctly updated the context. The confidence is achieved by either sending the header information more than once (optimistic approach) or by an ACK feedback from the decompressor. The optimistic approach is only used in the unidirectional mode and bidirectional optimistic mode. The number of times the same header information is (re)transmitted in the optimistic approach is an implementation-dependent variable.

A backward state transition occurs when one of the timeout counters are reset, a NACK or STATIC NACK feedback is received from the decompressor, or the packet to be compressed encounters a change in the static fields or the pattern of the dynamic fields. The two timeout counters are used to transit to IR or FO states and they are implementation-dependent variables. The timeout counters are only used in the unidirectional mode and the bidirectional optimistic mode.

IR state: It is used to initialize or to recover the context of the decompressor's static and dynamic fields. In this state, the compressor sends the entire header information by IR packets. The compressor stays in this state until it is fairly confident that the decompressor has established or updated the context correctly.

FO state: The purpose of this state is to effectively communicate the irregularities in the packet headers especially related to the dynamic parts of the context. The compressor transits to and from the FO state by forward/backward transitions discussed earlier, if there are changes in the static context. It enters this state from the IR state after transmitting static context or from the SO state after changes occur in a dynamic field (non-conformity of the previous pattern). FO packets carry context-updating information and are therefore protected by a CRC.

SO state: The compressor transits to this state when compression is optimal, and the compressor is sufficiently confident that the decompressor has all the context fields including the parameters of functions from SN to derive fields like IP-ID and TS to ensure correct decompression. In this state only a few bits are required for TS/IP-ID to be transmitted. The compressor changes from the SO to the IR or FO state with forward/backward transitions discussed earlier when the header pattern changes and it can no longer be compressed with present context. SO state packets are also protected by CRC.

3.6 Decompressor States

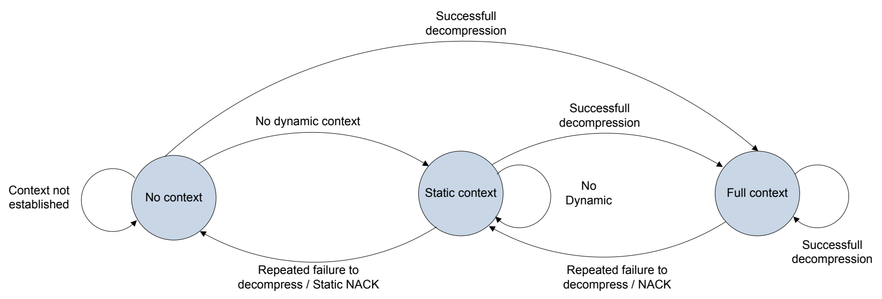

Successfull decompression
No dynamic context
Successfull decompression
Context not established
No context
Static context
No Dynamic
Full context
Successfull decompression
Repeated failure to decompress / Static NACK
Repeated failure to decompress / NACK

Figure 3.5: Decompressor state machine

The decompressor has three states as shown in Figure 3.5 which are independent of the modes: No context, static context and full context. It starts from the lowest compression state, no context, and gradually transits to higher order states.

No context: The decompressor will be in this state at the start of transmission when the decompressor context has not yet been established. In this state only IR packets are allowed to be received. Only after a successful decompression involving a CRC check the decompressor transits to the full context state.

Static context: In this state the decompressor is assumed to have the static information but not the dynamic information or it may have changed. In this state only IR, IR-DYN, UOR-2 packets described in section 3.9, are decompressed (carrying either an 8 bit or 7 bit CRC). Transitions to No context state from this state are triggered by repeated errors (unidirectional mode) or NACK/Static-NACK feedback (bidirectional mode).

Full context: Successful reception of any packet in the FO state of the compressor enables a transition to Full context. In this state, the decompressor has full context information. Once it enters this state it never leaves unless there are repeated unsuccessful decompression attempts (failure of CRC check, in unidirectional mode) or NACK/STATIC-NACK feedback (in bidirectional mode). Only then does it transit to the static context.

3.7 Encoding Methods

There are several encoding methods as specified in the RoHC framework employed for different dynamic fields of an IP/UDP/RTP header stack. These are:

3.7.1 LSB Encoding

As the name indicates, the LSB encoding is used to transmit the change that occurs in the least significant bits of a field when compared to a reference value stored in the context. The decompressor derives the original value from the received LSBs and the previously received reference value ($vref$) such that the value falls within the interpretation interval as defined by the function given in equation 3.1. The k received LSBs are enough to uniquely define the value within the interpretation interval.

$$f(vref, k) = [vref - p, vref + (2^k - 1) - p]. \tag{3.1}$$

$$|\overbrace{\text{- -}}^{\text{Interpretation Interval}}|$$

$$vref\ -\ p \qquad\qquad\qquad\qquad vref\ +\ (2^k - 1)\ -\ p$$

The selection function is referred to as $g(vref, v)$ such that $k = g(vref, v)$.

Here p is an integer used to shift the interpretation interval depending on the typical behaviour of different header fields like IP-ID, RTP-SN, or RTP-TS.

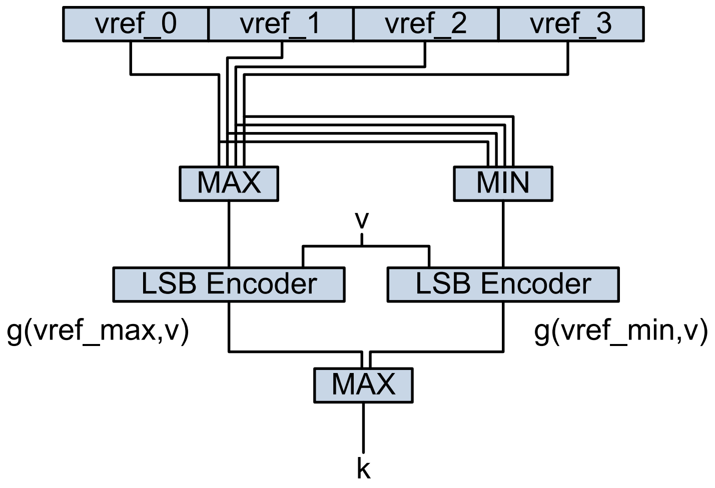


Figure 3.6: WLSB encoder

Window-based LSB (WLSB) Encoding

The compressor may not be able to determine the exact reference value at the decompressor side. Therefore, it maintains a sliding-window for candidates of reference values ranging from $vref_{min}$ to $vref_{max}$. The compressor chooses the maximum number of bits(k) to uniquely identify a value v using the equation:

$$k = max(g(vref_{min}, v), g(vref_{max}, v)). \tag{3.2}$$

The sliding-window is advanced by either a CRC confirmation, in case of U or O mode, and ACK, in case of R mode. WLSB modifies LSB encoding to achieve robustness by calculating k for a window of the stored field values and selecting the appropriate k. Figure 3.6 shows a block diagram of the WLSB method which explains how it is related to the LSB method.

3.7.2 Scaled RTP Timestamp Encoding

Common audio and video codecs have fixed sampling rates. For this reason and since an RTP packet transmits a constant number of these samples per unit time, the timestamp in successive RTP packets is usually increased by an integral multiple of some constant number called *ts_stride*. This property can be used to efficiently compress the timestamp of an RTP packet by sending a downscaled timestamp. The scaled timestamp can be expressed using less number of bits than the original timestamp and hence a first order compression can be achieved. A second order compression can be achieved by compressing the scaled timestamp using the WLSB encoding method discussed earlier.

The down-scaling of the timestamp follows equation 3.3 below:

$$ts = ts\_scaled \times ts\_stride + ts\_offset. \tag{3.3}$$

The *ts_stride* is communicated explicitly and *ts_offset* is communicated implicitly, as discussed in the following stages.

- **During initialization:** The compressor sends *ts_stride* and absolute value of *ts*. The decompressor initializes *ts_offset* using (*ts* modulo *ts_stride*).
- **During compression:** Only WLSB or Timer-based encoded ts_scaled is sent.
- **During decompression:** Original RTP-TS value is calculated using compressed *ts_scaled* and *ts_stride*, which are communicated at initialization using equation 3.3.
- **At wraparound:** The compressor does not reinitialize *ts_offset* at wraparound but the decompressor detects wraparound of unscaled *ts* and updates *ts_offset*.

3.7.3 Timer-based RTP Timestamp Compression

With fixed sampling intervals employed by audio and video applications and packets that are transmitted in lockstep with the sampling, RTP-TS is increased by an integral multiple. It can also be approximated by a linear function of the time of day taking into account the delay jitter between the compressor and decompressor. By using the local clock the decompressor can obtain an approximation of *ts_scaled* in the header considering its arrival time. This method is discussed in detail in [1], but is not used in this implementation.

3.7.4 IP-ID Offset Encoding

For IPv4, the IP-ID will increase by the same amount as RTP-SN. Therefore, it is more efficient to compress only the IP-ID offset relative to RTP-SN by WLSB encoding with p=0, using the formula

$$Offset = IP\ ID - RTP\ SN. \tag{3.4}$$

At the decompressor side, the IP-ID is calculated from the reference header (correctly verified by CRC) using the formula

$$IP - ID = Offset\_ref + RTP\ SN\_ref. \tag{3.5}$$

Some stacks do not use a counter to generate IP-ID values but instead they are generated randomly using a pseudo-random number generator, in such case IP-ID offset encoding cannot be used.

Table 3.1: SVDL Encoding

First bit/bits	No. of octets	No. of bits transferred
1	1	7
10	2	14
110	3	21
111	4	29

3.7.5 Self-Describing Variable Length Code (SDVL) Encoding

Parameters like *ts_stride* and some others vary widely. For optimal transfer of such values SDVL is used to encode them. The first few bits of the first octet determine the number of octets used as shown in the Table 3.1.

3.8 CRC

Cyclic redundancy checks are used by RoHC as the primary means for detecting erroneous decompression, just as in many other communication systems. CRCs play an important role in providing means for verification, especially in U/O mode where feedback is absent. Incorrect context update and prevention of residual errors from lower layer are mitigated by the CRC information carried by the RoHC packets. This adds to the robustness of the system.

Uncompressed packets like IR and IR-DYN carry an 8-bit CRC computed based on all the RoHC packet fields excluding the payload. As these are context-updating packets a larger size of CRC assures the context validation. Also for feedback type 2, as discussed in section 3.9, an 8-bit CRC is used.

Compressed packets like UO-0 and UO-1 contain 3-bit CRCs and packets like UO-2 contain a 7-bit CRC. For the compressed packets CRCs are computed on STATIC fields and DYNAMIC header fields of the original IP packets. The 3-bit, 7-bit and 8-bit CRCs are computed using the following polynomials.

3-bit CRC: $C(x) = 1 + x + x^3$

7-bit CRC: $C(x) = 1 + x + x^2 + x^3 + x^6 + x^7$

8-bit CRC: $C(x) = 1 + x + x^2 + x^8$

More details about the polynomials and the CRC calculations can be found in [8].

3.9 Major RoHC Packet Types

The RoHC framework defines several kinds of packets; they are discussed in brief below.

1. **RoHC packets :**
   - **IR and IR-DYN packets:** These are uncompressed RoHC packets used to initialize/communicate the static and/or dynamic parts of the context. IR-DYN is used to initialize/communicate only the dynamic part of the context.
   - **Type 0 packets (Ex: UO-0):** These packets are of smallest size (1 byte), hence, the most efficient. They are used when the decompressor can derive the necessary fields as a function of RTP-SN.
   - **Type 1 packets (Ex: UO-1-*):** These packets are used when WLSB encoded bits of RTP-SN exceed those available for Type 0 or when the parameters to the SN functions for deriving the TS or IP-ID fields change.
   - **Type 2 packets (Ex: UO-2-*):** These packets are used when the WLSB encoded bits of the RTP-SN exceed those available for Type 0 and Type 1, or when the parameters to the SN functions for deriving the TS or IP-ID fields change.
2. **Extension Packets:** Extension packets are used to convey information other than what is contained by lower compressed packets (like UO-1, UOR-2) without having to send uncompressed packets (IR, IR-Dyn). There are three types of extensions used in combination with base packets of type 1 and 2.
3. **Feedback Packets:** For RoHC modes that use feedback, they are sent either in separate RoHC packets or piggybacked on forward packets from the decompressor to the compressor. There are two types of feedback packets - FEEDBACK 1 packets, which are used to send ACK, and FEEDBACK 2 packets, which are used to send NACK or STATIC-NACK along with mode information.

Table 3.2 shows the main properties of the RoHC packets such as the CRC type, the updating capability, and the number of bits that the packet can carry for each of SN, IP-ID, and TS. The update column in the table specifies whether the packet can update the context or not. Further information about the structure of the packets and their properties is given in details in RFC3095 [1].

Packet type	**SN**	**IP-ID-1**	**IP-ID-2**	**TS**	**Update context**	**CRC type**	**Static fields**
R-0	6	0	0	0	No	0	No
R-0-CRC	7	0	0	0	Yes	7	No
R-1-ID	6	5	0	0	No	0	No
R-1-TS	6	0	0	5	No	0	No
UO-0	4	0	0	0	Yes	3	No
UO-1-ID	4	5	0	0	Yes	3	No
UO-1-TS	4	0	0	5	Yes	3	No
UOR-2-ID	6	5	0	0	Yes	7	No
R-1-ID-EXT0	9	8	0	0	No	0	No
UO-1-ID-EXT0	7	8	0	0	Yes	3	No
UOR-2-TS	6	0	0	5	Yes	7	No
R-1-TS-EXT0	9	0	0	8	No	0	No
UOR-2-ID-EXT0	9	8	0	0	Yes	7	No
UOR-2-TS-EXT0	9	0	0	8	Yes	7	No
R-1-TS-EXT1	9	8	0	8	No	0	No
UO-1-ID-EXT1	7	8	0	8	Yes	3	No
UO-1-ID-EXT2	7	16	0	8	Yes	3	No
UOR-2-TS-EXT1	9	8	0	8	Yes	7	No
R-1-ID-EXT2	9	16	0	8	No	0	No
R-1-TS-EXT2	9	8	0	16	No	0	No
UOR-2-ID-EXT2	9	16	0	8	Yes	7	No
UOR-2-TS-EXT2	9	8	0	16	Yes	7	No
R-1-ID-EXT3	14	16	16	29	No	0	No
UO-1-ID-EXT3	12	16	16	29	Yes	3	No
UOR-2-ID-EXT3	14	16	16	29	Yes	7	No
R-1-TS-EXT3	14	16	16	32	No	0	No
UOR-2-TS-EXT3	14	16	16	32	Yes	7	No
UO-1	4	0	0	6	Yes	3	No
R-1	6	0	0	6	No	0	No
UOR-2	6	0	0	6	Yes	7	No
R-1-EXT0	9	0	0	9	No	0	No
UOR-2-EXT0	9	0	0	9	Yes	7	No
R-1-EXT1	9	0	0	17	No	0	No
UOR-2-EXT1	9	0	0	17	Yes	7	No
R-1-EXT2	9	0	0	25	No	0	No
UOR-2-EXT2	9	0	0	25	Yes	7	No
R-1-EXT3	14	0	0	32	No	0	No
UOR-2-EXT3	14	16	16	32	Yes	7	No
IR-DYN	16	16	16	32	Yes	8	No
IR	16	16	16	32	Yes	8	Yes

Table 3.2: RoHC packets and their capabilities

Chapter 4

Design of RoHC Hardware Accelerators

Here certain hardware accelerators are designed to be used to accelerate the RoHC algorithm and are chosen based on profiling results of RoHC software discussed in section 5.3. These accelerators are used for both HW-SW and Full HW implementation of RoHC.

4.1 CRC Hardware Accelerator

There are different methods of computing CRCs based on both software and hardware implementations. The choice between hardware and software methods is application-dependent and is based on parameters such as speed, throughput, memory usage, and design time. These methods along with their advantages and disadvantages are tabulated in the Table 4.1.

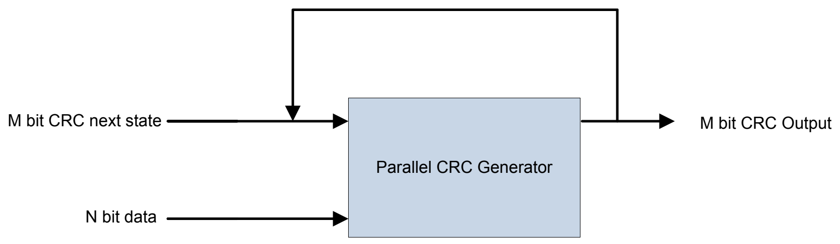


Figure 4.1: Parallel CRC

CRC computation requires bit level manipulations which are more efficient and fast in hardware, therefore CRC blocks are partitioned into hardware accelerators. For the

Table 4.1: Different CRC Methods

CRC Implementations	**Remarks**	**Advantages**	**Disadvantages**
Software - based, naive add-shift method	Software-based add-shift method emulates LFSR as shown in [8]	Quick design and prototyping time.	Large execution time and inefficient for real-time data
Software - based CRC table method	Data is used to index the CRCs which are pre-calculated and stored in a table	Faster execution time and greater throughput.	For an N-bit input and an M-bit CRC a large look-up table of size $M \times 2^N$ must be stored in the memory
Hardware - based LFSR method	Shift registers with linear feedback from XOR gates are used as per the polynomial to construct the CRC	Quick Design and prototyping time.	Low throughput and a latency of n clock cycles for calculation of n-bit CRC
Hardware - based parallel CRC method	CRC is calculated on data in parallel by performing necessary logic operations and by emulating the state transitions in the LFSR with the incoming parallel data as shown in Figure 4.1	Fastest CRC method with a throughput of 1 CRC per clock cycle.	More design and prototyping time.

RoHC system, the hardware based parallel CRC method is chosen as this gives the highest throughput. This is also experimentally proven on a NIOS II-based system with a sample data set of 50 bytes. The execution time for different CRC methods on a NIOS-based system is as plotted in Figure 4.2, and it can be seen that the hardware-based parallel CRC method has the fastest execution time.

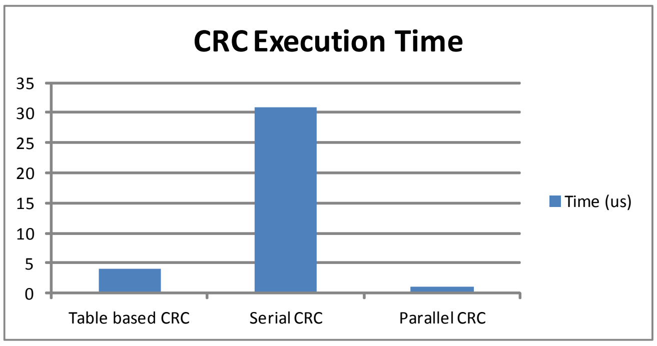


Figure 4.2: Execution time of different CRCs

In conclusion, we use parallel CRCs as designed by a method described in section 4.1.1, for both the HW-SW case and the full-HW case. However, in the case of a HW-SW parallel CRC block, the hardware is used in tandem with software to calculate the CRC of different data widths ranging from 32 bits to 8 bits as described in section 5.9.1.

Table 4.2: H1 Matrix

$crc\_out = f_{crc}(crc\_in = 0, din)$	$crc\_out[0]$	$crc\_out[1]$	$crc\_out[2]$	
$din[0]$	1	1	0	$din[0] = 1$
$din[1]$	0	0	1	$din[1] = 1$
$din[2]$	0	1	0	$din[2] = 1$
$din[3]$	1	0	0	$din[3] = 1$
$din[4]$	1	0	1	$din[4] = 1$
$din[5]$	1	1	1	$din[5] = 1$
$din[6]$	0	1	1	$din[6] = 1$
$din[7]$	1	1	0	$din[7] = 1$

Table 4.3: H2 Matrix

$crc\_out = f_{crc}(crc\_in, din = 0)$	$crc\_out[0]$	$crc\_out[1]$	$crc\_out[2]$	
$crc\_in[0]$	1	1	0	$crc\_in[0] = 1$
$crc\_in[1]$	0	0	1	$crc\_in[1] = 1$
$crc\_in[2]$	0	1	0	$crc\_in[2] = 1$

4.1.1 Design of Parallel CRCs

The design of a parallel CRC block is done as per the method proposed in [9], which leverages the serial CRC generator and discrete-time linear-system properties of CRCs to generate parallel CRCs.The parallel CRC generator block is as shown in Figure 4.1. The design process involves the following steps:

1. First a C-based serial CRC simulator is implemented. The logic works on 8-bit data and CRC initialization values to produce an M-bit CRC. For 16 and 32 bit data, the same logic is used in cascade iteratively 2 or 4 times, respectively. And the CRC initialization value being updated with the 8-bit CRC output with each iteration.

2. Then an H1 matrix is formed based on the equation $crc\_out = f_{crc}(crc\_in = 0, din)$ for 1-shot high values of din like 1, 2, 4, 8, 16..., and neglecting crc_in, using the CRC simulator developed in 1.

3. Then an H2 matrix is formed based on the equation $crc\_out = f_{crc}(crc\_in, din = 0)$ for 1-shot high values of crc_in like 1, 2, 4, 8, 16..., and neglecting din, using the CRC simulator developed in 1.

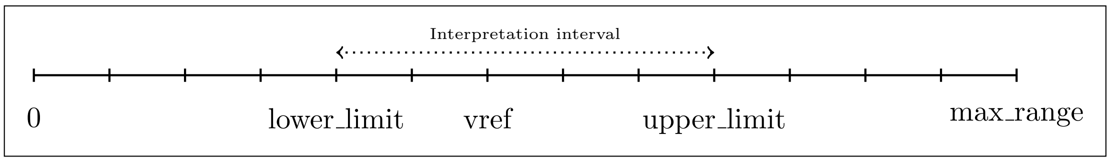
4. Then the logical equations from the H1 and H2 matrices are combined by logical modulo-2 arithmetic to generate parallel CRC equations for 8-bit, 16-bit and 32-bit data.

For example, the H1 and H2 matrices for 3-bit CRC $(1 + x + x^3)$ and 8-bit data are as shown in Table 4.2 and Table 4.3. The XOR equations for parallel CRCs based on the H1 and H2 matrices are as shown in equations 4.1.

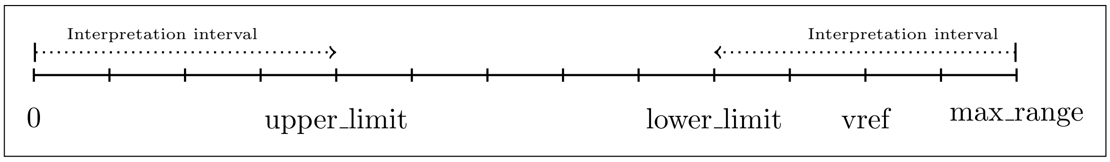
$$\begin{aligned}
crc\_out[0] &= \overbrace{crc\_in[0] }^{\text{H2 matrix}} \oplus \overbrace{din[0] \oplus din[3] \oplus din[4] \oplus din[5] \oplus din[7]}^{\text{H1 matrix}} \\
crc\_out[1] &= crc\_in[0] \oplus crc\_in[2] \oplus din[0] \oplus din[2] \oplus din[5] \oplus din[6] \oplus din[7] \\
crc\_out[2] &= crc\_in[1] \qquad\qquad\quad \oplus din[1] \oplus din[4] \oplus din[5] \oplus din[6]
\end{aligned} \tag{4.1}$$

4.2 Least Significant Bit Encoding (LSB Encoding)

As explained earlier in subsection 3.7.1, LSB encoding makes use of the interpretation interval technique to efficiently encode a field with respect to a reference value stored in the context. The decompression of the LSB bits is guaranteed if both the compressor and the decompressor use the same interpretation interval. An illustration of the interpretation interval is shown in Figure 4.3a below. Since the header fields have finite number of bits to describe their values, the interpretation interval might wrap around as shown in Figure 4.3b.



(a) Normal interpretation interval



(b) Wraparound interpretation interval

Figure 4.3: Different interpretation interval cases

The terms *upper_limit* and *lower_limit* are defined in equations 4.2 and 4.3 respectively as

$$upper\_limit = vref + 2^k - 1 - p \tag{4.2}$$

and

$$lower\_limit = vref - p. \tag{4.3}$$

In equation 4.2, k is defined as the least number of bits such that the field value lies inside the interpretation interval below:

$$lower\_limit \leq value \leq upper\_limit.$$

If the interpretation interval is wrapped around, $upper\_limit < lower\_limit$, then the condition is changed to :

$$lower\_limit \leq value \quad or \quad value \leq upper\_limit.$$

In equation 4.2, p is defined as the shifting value that shift the interpretation interval and it differs from one field to another. Below are the p values for the Sequence Number (SN), the Internet Protocol Identification (IP-ID), and the Time Stamp (TS) as specified in [1]:

$$p(sn) = \begin{cases} 1 & if k \leq 4 \\ 2^{k-5} - 1 & \text{otherwise} \end{cases}, \tag{4.4}$$

$$p(ip - id) = 0, \tag{4.5}$$

and

$$p(ts) = 2^{k-2} - 1. \tag{4.6}$$

To find k, equation 4.4, 4.5 and 4.6 parameterize the shifting value p as follows :

$$p = a2^{k-b} - c,$$

where a, b and c are shown in Table 4.4.

Table 4.4: Parametrization of the shifting value p

Field	**a**	**b**	**c**	**Condition**
SN	0	0	-1	if $k \leq 4$
	1	5	1	if $k > 4$
IP-ID	0	0	0	-
TS	1	2	1	-

By setting $value = upper\_limit$ and $value = lower\_limit$, k_1 and k_2 are found respectively. k_1 and k_2 are calculated in equations 4.7 and 4.8 respectively as

$$k_1 = \lceil log_2(R_{fd} - c + 1) + b - log_2(2^b - a) \rceil, \tag{4.7}$$

and

$$k_2 = \lceil log_2(R_{bk} + c) + log_2(2^b) \rceil. \tag{4.8}$$

To find the least number of bits, k is calculated as

$$k = min(k_1, k_2). \tag{4.9}$$

where

R_{fd} : is the absolute forward distance between $value$ and $vref$

and

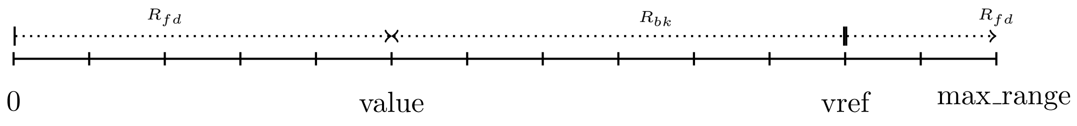
R_{bk} : is the absolute backward distance between $value$ and $vref$.

In [3], $k1$ is always assumed less than $k2$ and hence considered as the solution of k. In this work, it is shown that any of $k1$ or $k2$ can be a solution of k and the earlier assumption by [3] is not correct and cannot cover all cases.

Since p value is different from one field to another, some special cases are exist. For instance, the value of p for IP-ID is zero and there are two values of p for the SN. Since p is equal to zero for IP-ID, k is calculated from equation 4.7 only. Unlike [3], in which $k(sn)$ is always calculated first for $p = 1$, k(sn) is considered equal to 4 bits (the least number of bits that a RoHC packet can transmit) if $R_{fd} \leq 14$ or $R_{bk} \leq 1$. This will avoid an unnecessary iteration of trying different p values.

The rest of this section focuses on how to find which equation, $k1$ or $k2$, is the correct solution for k without the need to calculate both equations in an attempt to reduce both processing power and hardware. By looking at the position of $value$ and $vref$, two cases are found as follows:

4.2.1 Case 1 when ($value < vref$)



The least number of bits is calculated in equation 4.10 as

$$k = \begin{cases} k2 & \text{if } R_{bk} \leq R_{bk\_max} \\ k1 & \text{otherwise} \end{cases}, \tag{4.10}$$

where R_{bk_max} is the maximum backward distance. R_{bk_max} is calculated in equation 4.11 as

$$R_{bk\_max} = 2^{k_{max}-b} - c, \tag{4.11}$$

where k_{max} is defined as the maximum number of bits that can describe a field i.e. k_{max} is equal to 16 bits for both SN and IP-ID and it is equal to 32 bits for TS.

R_{bk} and R_{fd} are calculated from the graph as follows:

$$R_{bk} = vref - value, \tag{4.12}$$

and

$$R_{fd} = max\_range + value - vref + 1. \tag{4.13}$$

Thus,

$$R_{fd} = max\_range - R_{bk} + 1. \tag{4.14}$$

To prove equation 4.10, substituting equation 4.14 when $R_{bk} = R_{bk\_max}$ in equation 4.7 yields:

$$k_1 = log_2(2^{k_{max}} \cdot \frac{(2^b - 1)}{2^b} + 1) + b - log_2(2^b - 1) \qquad \text{(since a=1)}. \tag{4.15}$$

By underestimating

$$log_2(2^{k_{max}} \cdot \frac{(2^b - 1)}{2^b} + 1) \simeq log_2(2^{k_{max}} \cdot \frac{(2^b - 1)}{2^b}),$$

this yields

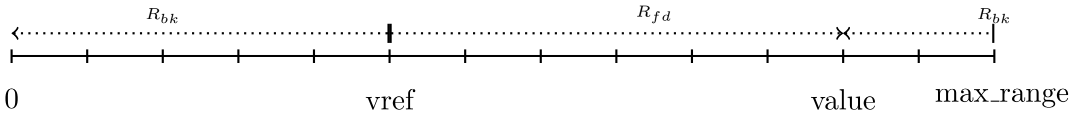
$$k_1 \simeq k_{max}.$$

By substituting $R_{bk} = R_{bk\_max}$ in equation 4.8

$$k_2 = k_{max}.$$

Since an underestimation of the first term in equation 4.15 is done, $k_2 < k_1$ when R_{bk} is less than or equal R_{bk_max}. Otherwise, $k1$ is less than $k2$. Therefore, using this condition there is no need to calculate both $k1$ and $k2$ in order to find k and the processing power is reduced by half.

4.2.2 Case 2 when ($value > vref$)



The least number of bits is calculated in equation 4.16 as

$$k = \begin{cases} k2 & \text{if } R_{bk} \leq R_{bk\_max} \\ k1 & \text{otherwise} \end{cases}, \tag{4.16}$$

where R_{fd} and R_{bk} are calculated as

$$R_{fd} = value - vref, \tag{4.17}$$

and

$$R_{bk} = max\_range + vref + 1 - value. \tag{4.18}$$

Similar to subsection 4.2.1, equation 4.16 can be proved by substituting equation 4.14 when $R_{bk} = R_{bk\_max}$ in equation 4.7 and $R_{bk} = R_{bk\_max}$ in equation 4.8.

4.2.3 Hardware Implementation of LSB Encoding

The hardware complexity of equation 4.7 and 4.8 resides in the logarithmic terms. Since K should be an integral number of bits, the log_2 implementation can be reduced to a Leading One Detector (LOD)for calculating the integer part of the logarithm.

The improvements in this implementation of the LSB encoder over the one presented in [3] include accounting for all possible cases with no extra hardware cost, introducing a zero-error implementation of the logarithmic equations without the need for floating point logarithm hardware, and reducing the need of calculating k(SN) using two shifting values p.

In order to find the value of k in [3], the result of equation 4.7 is ceiled only if the integer remainder of the first logarithmic term $(R_{fd} - c + 1)$is larger than the integer remainder of the second logarithmic term $(2^b - a)$. Using this method, an error of positive one might be induced to k. Although sending an extra bit will not lead to a decompression failure, it definitely influences the decision of which packet to send. Sending a different packet rather than the optimal one might mean sending more bits for other fields as well leading to an inefficient compression.

The error induced from using the ceiling operation on the integer remainder comparison in [3] is related to the fact that the integer remainders might describe fractions on different binary logarithmic scale. In other words, if the integer parts of both logarithms are not equal, the comparison is invalid. This is a very typical case.

In this work, a zero-error LSB function is proposed by adding an extra step to the integer remainder comparison. The remainder of one term should first be shifted to the same binary logarithmic scale as the other, and then the ceiling is done based on the comparison. The example below shows the difference between the proposed method and the original in [3].

Example:
Assume the first and the second logarithmic terms in equation 4.7 are 1058 and 31, as shown below:

$$(1058)_2 = 10000100010 \qquad floor(log_2(1058)) = 10$$
$$(31)_2 = 11111 \qquad floor(log_2(31)) = 4$$

1. Using the integer remainder comparison method as in [3],

$$remainder_1 = 34 \qquad remainder_2 = 15$$

$$k = 10{+}5{-}4{+}1(\text{since } remainder_1 > remainder_2) \quad \Longrightarrow \quad k = 12 \text{ (k should be 11)}$$

2. Using the zero-error proposed method

$$\begin{aligned} remainder_2 &= 15 \times (\lfloor log_2(1058)\rfloor) - \lfloor(log_2(31)\rfloor) \\ &= 960 \end{aligned}$$

$$k = 10 + 5 - 4 + 0(\text{since } 34 < 960)) \quad k = 11(correct).$$

Below is a mathematical proof that the error is always zero in this approach.

Any positive integer number N can be described in its binary form as:

$$N = \sum_{i=0}^{x} 2^i.N(i). \tag{4.19}$$

$$N = 2^x + \sum_{i=0}^{x-1} 2^i.N(i). \tag{4.20}$$

$$\text{Let } r = \sum_{i=0}^{x-1} 2^i.N(i). \tag{4.21}$$

$$N = 2^x + r \tag{4.22}$$

$$log_2(N) = x + log_2(2^x + r) - \lfloor log_2(2^x + r)\rfloor. \tag{4.23}$$

$$\text{Let } R = log_2(2^x + r) - \lfloor log_2(2^x + r)\rfloor. \tag{4.24}$$

In the proposed zero-error comparison method, the integer remainder of one of the logarithmic terms ($N1$) is shifted to same binary logarithmic scale of the second term ($N2$). The value of the shifting (n) is the difference between the integer results of the two logarithmic terms i.e. $n = x1 - x2$. $R^{'}$ is defined as the shifted integer remainder and it is calculated as in the equation bellow:

$$\text{Let } R^{'} = log_2\{2^n \cdot (2^x + r)\} - \lfloor log_2\{2^n \cdot (2^x + r)\}\rfloor. \tag{4.25}$$

If the error of shifting the remainder is zero, R must equal $R^{'}$

$$\begin{aligned} R^{'} &= log_2\{2^n \cdot (2^x + r)\} - \lfloor log_2\{2^n \cdot (2^x + r)\}\rfloor \\ &= n + log_2\{(2^x + r)\} - n - \lfloor log_2\{2^n \cdot (2^x + r)\}\rfloor \\ &= log_2(2^x + r) - \lfloor log_2(2^x + r)\rfloor \\ \text{Thus } R^{'} &= R \end{aligned}$$

Figure 4.4 shows a block diagram of a common hardware for calculating either $k1$ or $k2$ equation. After finding which one of the equations is the solution of k, all of the parameters shown in Figure 4.2 - X, Y, $R1$ and $R2$ - are calculated and sent to the LSB hardware to calculate the final k. X, Y, $R1$ and $R2$ are inner terms in equations 4.7 and 4.8 as follows:

$$K_1 = \lceil log_2(\overbrace{R_{fd} - c + 1}^{2^X + R1}) + b - log_2(\overbrace{2^b - a}^{2^Y + R2})\rceil \tag{4.26}$$

$$K_2 = \lceil log_2(\overbrace{R_{bk} + c}^{2^X + R1}) + log_2(\overbrace{\;2^b\;}^{2^Y + R2})\rceil \tag{4.27}$$

The hardware architecture of the LSB encoder consists of the following components:

1. **Leading-one detector (LOD)**: The LOD is implemented to emulate the integer part of the log_2 function. This component detects the leading one from the MSB of the data.
2. **Encoder**: The encoder outputs Y (constant value) as a function of a, b, and c as earlier defined in this section.
3. **Shifter**: the shifter is used to shift the integer remainder $R2$ to the same binary logarithmic scale of the integer remainder $R1$.
4. **Basic components**: Adders, MUXes, comparators

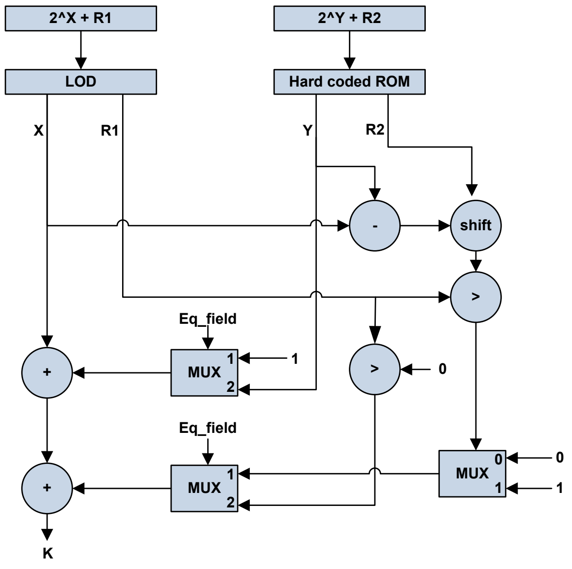


Figure 4.4: LSB encoder

The number of LOD components needed for the LSB encoding method is only one since only one equation is needed for the calculation and the second logarithmic term in the equations is a pre-known constant. The LSB encoder occupies 257 ALUT when it is synthesised on Altera Arria II GX FPGA and can run with a frequency of 150 MHz.

Chapter 5

HW-SW Solution

5.1 Introduction

A hardware-software co-design methodology is adopted to evaluate the performance enhancement that can be achieved by using hardware accelerators. This methodology will provide a method for evaluating potential future off-loading of LTE Layer 2 (PDCP) packet-data processing to hardware accelerators. It will also help in evaluating system-level design trade-offs such as determining if hardware acceleration of certain parts of RoHC can cater to the increasing capacity needs (taking into account the maximum capacity that is possible to support in LTE) or if full hardware-based RoHC is needed. In this chapter, we discuss the hardware-software co-design implementation of RoHC and also present the results of this methodology in section 5.11.

The reference code used in this case is open source RoHC libraries implemented as defined in the RFC 3095 spec [1] by the IETF. More information about the reference code and its features is given in [10].

The requirements for software and hardware partitioning of this co-design framework are based on profiling of the reference code based on worst-case execution. The concurrency in the RoHC algorithm can be exploited when looking for synergies between hardware and software. The features of this hardware-software design are given in Table 5.1. The RoHC libraries are ported to Alteras NIOS II softcore processors Software Build Tools (SBT) are used for software design. Certain functionality that can be executed in parallel is implemented in hardware for increasing the performance of the system and this is determined by profiling of the RoHC libraries.

Table 5.1: HW-SW implementation features

Feature	**Status**
Mode	U,O and R
Profile	1
Feedback processing	Supported
IP level	2
IPv4	Supported
IPv6	Supported
Compression list	Not supported

5.2 Porting of RoHC to FPGA

The GCC RoHC libraries, as mentioned in [10], are used as a reference model and are ported to a NIOS II-based embedded system. This gives us a common platform for profiling and capturing the performance metrics. Porting of the libraries was done in the following steps:

1. Isolation of the compressor and decompressor along with their libraries and dependencies. Removing of shared dependencies between the compressor and decompressor.

2. Inclusion of feedback functions into the compressor and compressor-specific header files.

3. Optimizing with respect to profile 1. Other profiles and their specific dependencies are removed and only profile 1 (IP/UDP/RTP) for VoIP application is emphasized. Several functions are rewritten with respect to profile 1 only.

4. List compression for IPv6 extension headers and the related functions are removed keeping in mind the requirements to concentrate on profile1 for VoIP application.

5. Certain GCC headers like netinet/ethernet.h, netinet/ip.h, netinet/ip6.h, netinet/ udp.h, and netinet/in.h are removed as NIOS II SBT system does not support them. The definitions and their respective (network) functions are rewritten in a custom header file.

6. Dependency on PCAP libraries for reading and writing of packets is removed and instead the test packets are stored in memory as constant arrays.

7. Several functions are optimized and rewritten to increase the performance.

8. Many unnecessary measurements of statistical parameters are removed from the code.

In summary, all these measures are taken to efficiently port the RoHC libraries to the NIOS II-based system to be run on the Altera FPGA board in order to yield optimum performance. The function hierarchy after porting the RoHC library is as shown in Figure 5.1

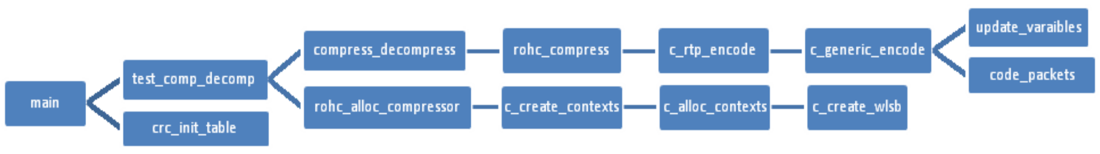


Figure 5.1: Software function tree

5.3 Profiling

To degree of partitioning of the RoHC functionality into hardware and software is determined by software profiling of the RoHC reference code from [10]. The two approaches used for profiling are the following.

5.3.1 Valgrid

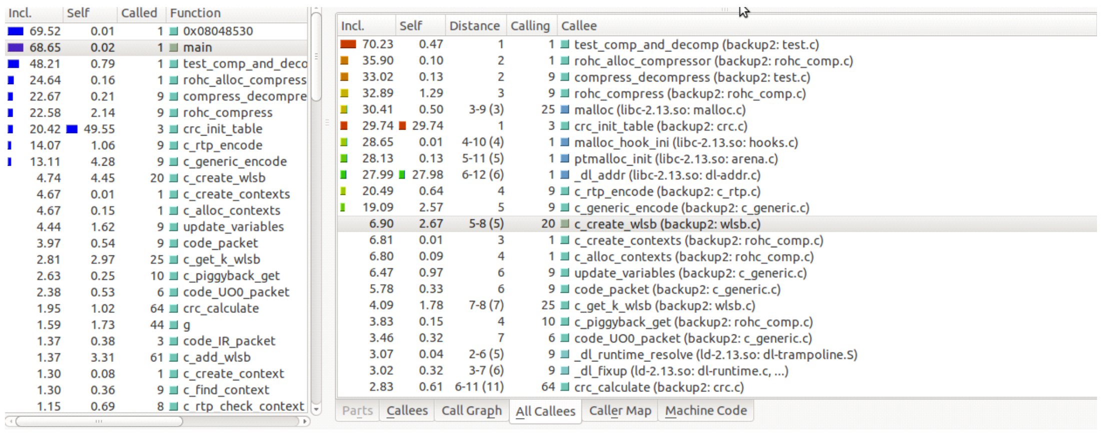
This is a runtime instrumentation framework with tools like memcheck, cachegrind (cache and branch prediction profiler),and callgrind (call-graph generation profiler).

Valgrid is used to make an initial estimate of worst-case execution times of functions by approximate profiling of memory footprint/cache and function-call profiling based on flat profile data (event counts such as data reads, cache misses, etc.) of the GCC-based RoHC libraries. As shown in Figure 5.2, the metric of self cost multiplied by the called fields determines the bottleneck in the reference code. This turns out to be the CRC and WLSB functions.

5.3.2 Altera based Performance Counters

Due to lack of good tools for low-resolution timing profiling, Valgrid is used to estimate the profiling data. However, after porting the RoHC libraries to the NIOS II SBT and running on FPGA, performance counters are included in the NIOS-based embedded system to measure the execution time of different sections of the code. Each performance counter keeps track of execution time of each section of the code and the number of occurrences. This runtime performance measurement gives us accurate profiling data identifying the bottlenecks.

The functions identified as bottlenecks by profiling are recognized as possible candidates for hardware implementation to increase the performance. The components identified



Incl.	Self	Called	Function
69.52	0.01	1	0x08048530
68.65	0.02	1	main
48.21	0.79	1	test_comp_and_deco
24.64	0.16	1	rohc_alloc_compress
22.67	0.21	9	compress_decompre
22.58	2.14	9	rohc_compress
20.42	49.55	3	crc_init_table
14.07	1.06	9	c_rtp_encode
13.11	4.28	9	c_generic_encode
4.74	4.45	20	c_create_wlsb
4.67	0.01	1	c_create_contexts
4.67	0.15	1	c_alloc_contexts
4.44	1.62	9	update_variables
3.97	0.54	9	code_packet
2.81	2.97	25	c_get_k_wlsb
2.63	0.25	10	c_piggyback_get
2.38	0.53	6	code_UO0_packet
1.95	1.02	64	crc_calculate
1.59	1.73	44	g
1.37	0.38	3	code_IR_packet
1.37	3.31	61	c_add_wlsb
1.30	0.08	1	c_create_context
1.30	0.36	9	c_find_context
1.15	0.69	8	c_rtp_check_context

Incl.	Self	Distance	Calling	Callee
70.23	0.47	1	1	test_comp_and_decomp (backup2: test.c)
35.90	0.10	2	1	rohc_alloc_compressor (backup2: rohc_comp.c)
33.02	0.13	2	9	compress_decompress (backup2: test.c)
32.89	1.29	3	9	rohc_compress (backup2: rohc_comp.c)
30.41	0.50	3-9 (3)	25	malloc (libc-2.13.so: malloc.c)
29.74	29.74	1	3	crc_init_table (backup2: crc.c)
28.65	0.01	4-10 (4)	1	malloc_hook_ini (libc-2.13.so: hooks.c)
28.13	0.13	5-11 (5)	1	ptmalloc_init (libc-2.13.so: arena.c)
27.99	27.98	6-12 (6)	1	_dl_addr (libc-2.13.so: dl-addr.c)
20.49	0.64	4	9	c_rtp_encode (backup2: c_rtp.c)
19.09	2.57	5	9	c_generic_encode (backup2: c_generic.c)
6.90	2.67	5-8 (5)	20	c_create_wlsb (backup2: wlsb.c)
6.81	0.01	3	1	c_create_contexts (backup2: rohc_comp.c)
6.80	0.09	4	1	c_alloc_contexts (backup2: rohc_comp.c)
6.47	0.97	6	9	update_variables (backup2: c_generic.c)
5.78	0.33	6	9	code_packet (backup2: c_generic.c)
4.09	1.78	7-8 (7)	25	c_get_k_wlsb (backup2: wlsb.c)
3.83	0.15	4	10	c_piggyback_get (backup2: rohc_comp.c)
3.46	0.32	7	6	code_UO0_packet (backup2: c_generic.c)
3.07	0.04	2-6 (5)	9	_dl_runtime_resolve (ld-2.13.so: dl-trampoline.S)
3.02	0.32	3-7 (6)	9	_dl_fixup (ld-2.13.so: dl-runtime.c, ...)
2.83	0.61	6-11 (11)	64	crc_calculate (backup2: crc.c)

Figure 5.2: Profiling using Valgrid

after profiling are CRC and WLSB because of their computational cost and frequency of use in a RoHC system.

5.4 Algorithmic Partitioning between Hardware and Software

The hardware-software partitioning is done based on the profiling data, which generate partitioning metrics like:

1. **Computation cost**: determines how computationally intensive the functionality/component is.
2. **Communication cost**: determines if it is beneficial to implement the functionality/component as hardware, taking into account the cost of communicating the data to the accelerator.
3. **Frequency of use cost**: determines the frequency of usage of the particular hardware functionality/ component.

The RoHC functionality is captured in the flow chart as shown in Figure 5.3. Each functionality is implemented as a sequence of functions as shown in Figure 5.1, interleaved efficiently with data structures containing packet data, context data and some necessary statistics. Hardware implementation of WLSB encoding and CRC algorithms is determined by profiling to be beneficial to boost the performance of the RoHC software libraries. These hardware accelerators are either implemented as custom logic components or custom instructions is a Nios II-based embedded system for RoHC.

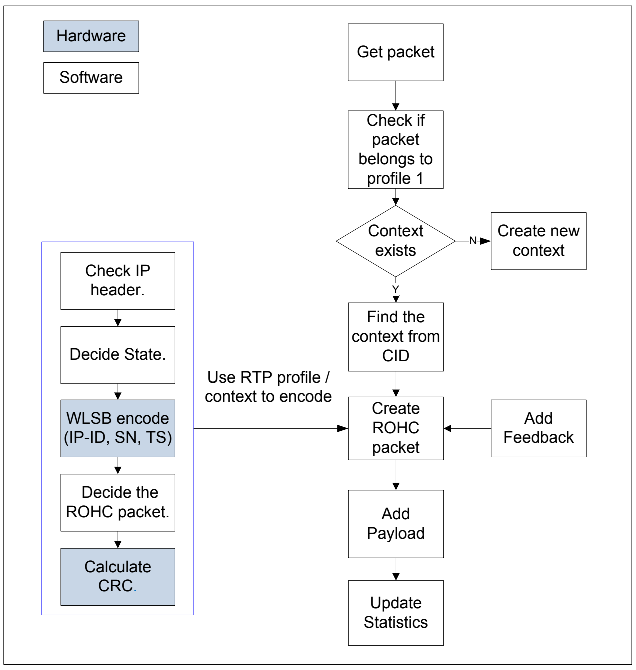


Figure 5.3: RoHC algorithmic partitioning

5.5 HW-SW co-design flow for a RoHC system

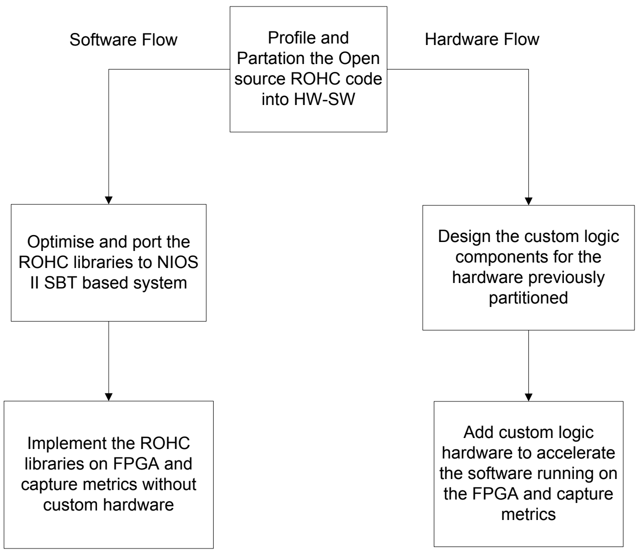

Software Flow
Profile and Partation the Open source ROHC code into HW-SW
Hardware Flow
Optimise and port the ROHC libraries to NIOS II SBT based system
Design the custom logic components for the hardware previously partitioned
Implement the ROHC libraries on FPGA and capture metrics without custom hardware
Add custom logic hardware to accelerate the software running on the FPGA and capture metrics

Figure 5.4: HW-SW co-design flow for a RoHC system

The hardware-software co-design for the RoHC system is as shown in Figure 5.4 consists of

1. **Software flow**: Here the RoHC libraries are optimized and ported to NIOS II SBT to be run on FPGA.
2. **Hardware flow**: Here the hardware components (WLSB and CRC) are designed to accelerate the performance of software RoHC libraries.

Finally, cross-verification is used to check the functionality of the RoHC system consisting of the interdependent software libraries and their hardware accelerators after they have been individually tested.

5.6 RoHC Embedded System

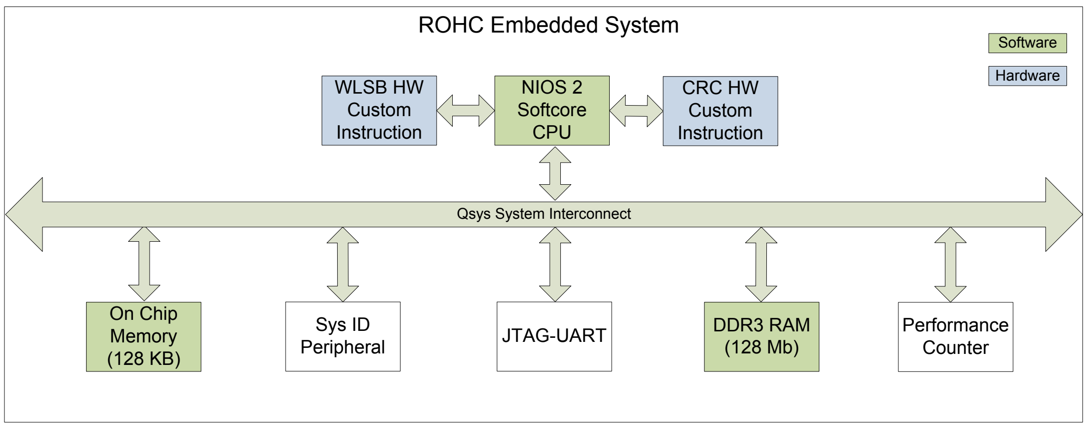


Figure 5.5: RoHC embedded system

The embedded system for RoHC is assembled using Qsys of Altera with Altera's intellectual property (such as performance counters, Sys ID peripheral, JTAG UART), memories (On-chip and DDR3 RAM) and custom logic components (CRC , WLSB) interconnected by Qsys interconnect is as shown in Figure 5.5.

1. **Software blocks**: A Nios II softcore processor of type F with 4 kB instruction and 32 kB data cache is used along with hardware support for division and multiplication (DSP blocks) to execute the RoHC code which resides in the on-chip memory (128 kB). As the number of users increase to support high capacity at the base station side, maintaining their context data in on-chip memory is not possible. Therefore, DDR3 RAM was added.

2. **Hardware blocks**: These are the hardware accelerators designed to increase the performance of the RoHC code running in the NIOS II processor. These hardware accelerators can be implemented either as custom components or custom instructions, which are further discussed in 5.8. In this case they are implemented as custom instructions for the Nios II processor.

3. **Other peripherals**:

   JTAG UART: This is an Altera IP which provides means to communicate with a host PC via serial character streams between the host and the Qsys system. It is mainly used for debugging purposes in the Qsys system.

   Sys ID Peripheral: This is an Altera-based peripheral which assigns the Qsys system a unique ID and timestamps. The NIOS II IDE verifies the system ID before downloading new software to the system. This is to verify that the software runs on a Qsys System for which it is written and compiled.

   Performance Counters: This is a block of counters which can measure the execution time of selected code by keeping track of time and occurrences of that section of code. This is used to measure the performance of the RoHC system.

   The device drivers to the above-mentioned peripheral and standard interfaces are provided by Alteras HAL (Hardware-Abstraction Layer). The NIOS software build tools generate a HAL board-support package from the configuration of our Qsys system.

5.7 Qsys System Interconnect

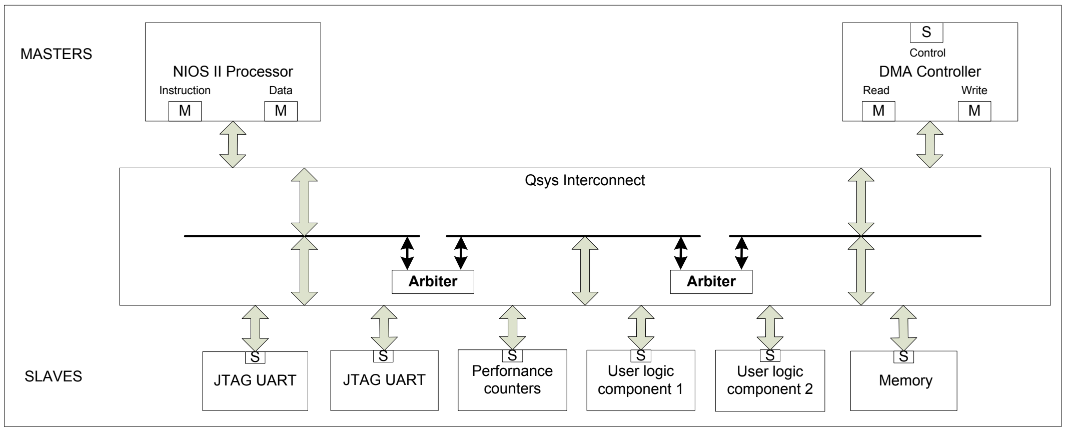


Figure 5.6: Qsys system interconnect

The Qsys interconnect is a high-bandwidth structure for interconnecting the components with Avalon interfaces. There are two types of Avalon interfaces:

1. **Avalon Streaming (ST)**: Connects sources and sinks (components) in a unidirectional data stream.
2. **Avalon Memory Maped (MM)**: This is an address-based interface which connects the master and slave components that communicate using write and read commands in a memory-mapped fashion.

The Avalon MM implementation of Qsys interconnect is as shown in Figure 5.6, The M interface signifies masters and S signifies slaves. The Avalon masters, the NIOS 2 processor and DMA controller, are connected to Avalon slaves by Qsys interconnect. This Qsys interconnect is based on a network-on-chip architecture where transactions between source and sink are encapsulated in packets [11]. MM read and write protocols and signal mappings are as discussed in [11].

The Qsys interconnect takes care of arbitration, address decoding, data-path multiplexing and interrupts but the most important function of Qsys is that it eliminates the need to create arbitration hardware manually. It allows multiple master interfaces to transfer data to independent slaves simultaneously by slave-side arbitration, unlike traditional host-side arbitration where each master should wait until it is granted access to the shared bus. The arbitration logic stalls the master interface only when multiple masters attempt to access the same slave simultaneously. Hence, there is no unnecessary master-slave contention.

5.8 Custom Logic Instructions vs Custom Logic Peripherals

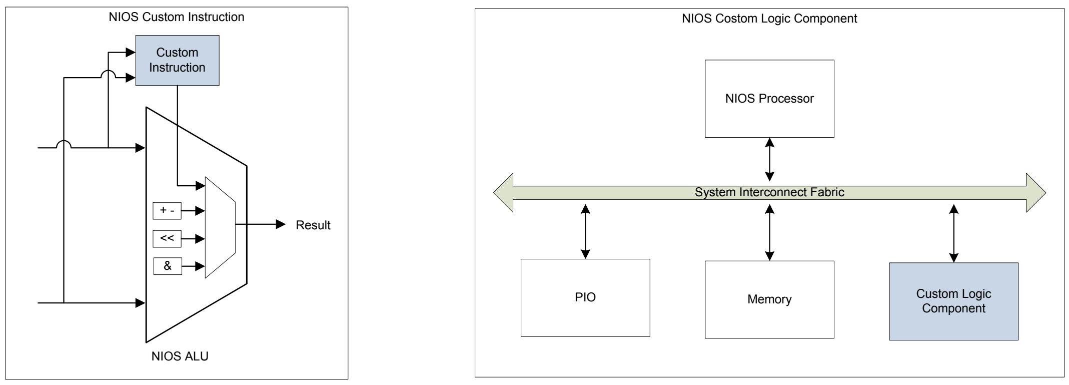


Figure 5.7: Custom logic instructions vs custom logic component

User-logic hardware blocks can be implemented either as custom-logic instructions or custom-logic component. HW acceleration has two benefits. Hardware is much faster and more efficient than software, and implementing functionality in hardware frees up the processor to perform other functions in parallel.

Custom Logic component:These are user-hardware implementations which can be included in the build of the Qsys System, as shown in Figure 5.7. In a memory-mapped system they act as peripherals where the masters (NIOS II System) can write data into them and results can be read from them, either in single or multiple clock cycles.

NIOS II Custom instructions: These are user-defined instructions which are implemented in custom-logic blocks adjacent to the ALU in the data-path of the NIOS II processor, as shown in Figure 5.7. They consist of custom-logic blocks, which are designed similarly to custom-logic peripherals (either as single-clock cycle or multi-clock cycle components), and software macros which are used to access the custom-logic block.

The parallel CRC algorithm was implemented both as custom instruction and custom component and tested for a sample data set. The results are plotted in the form of a bar graph in Figure 5.8. This shows that a custom instruction is almost twice as fast as a custom peripheral. This holds for both the case when there is data in external memory and the case when the data is in on-chip memory. The higher speed of custom instructions is attributed to the lack of arbitration, address decoding, and FIFO operations, which are all needed in the case of memory-mapped custom-logic peripherals.

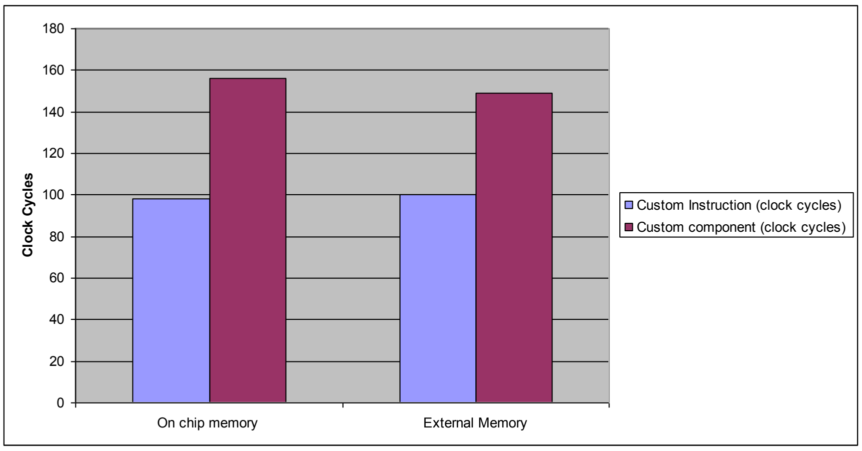


Figure 5.8: Performance of custom logic instructions vs. custom logic component

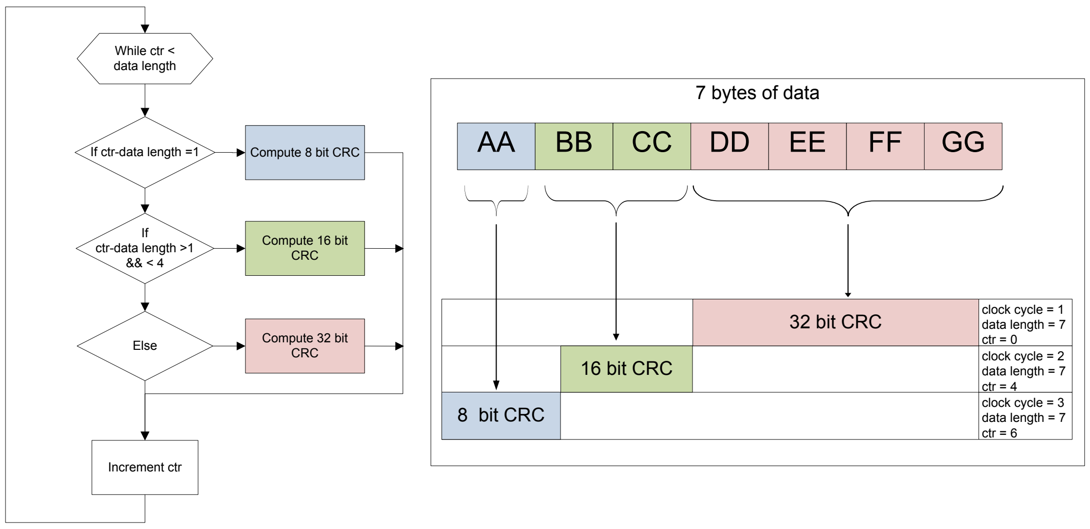


Figure 5.9: Mixed data-with CRC algorithm

5.9 Hardware Accelerators in RoHC System

5.9.1 Mixed data-width CRC Hardware Accelerator

In the RoHC software model the data is arranged in CRC buffers in 32 bit tiles in the memory. It is not known that the data for which a CRC is going to be computed will fit exactly in these 32-bit tiles. Zero padding is not an option since it changes the CRC all together. Therefore, a 32-bit parallel CRC generator is not enough when implementing CRC for RoHC, which is the problem.

One solution to this problem is to have a dynamic selection of parallel CRC generators for 32 bit as well as 16 bit and 8 bit data formats. Then logic is used to combine these mixed data-width CRCs inorder to compute CRC, irrespective of whether the data end in 32-bit tiles or not.

The logical flow chart for this kind of CRC computation and an example illustration is as shown in Figure 5.9. The CRC is calculated in parallel over a data size of 7 bytes, which includes first the computation of a 32-bit CRC over the first 4 bytes, then a 16-bit CRC, and finally an 8-bit CRC for the last byte.

5.9.2 CRC Hardware Accelerator Architecture

Due to the better performance of custom instructions over custom-logic components, as discussed in section 5.8, we choose to implement the CRC hardware accelerator as a NIOS II custom instruction. The advantage of working with NIOS II embedded system

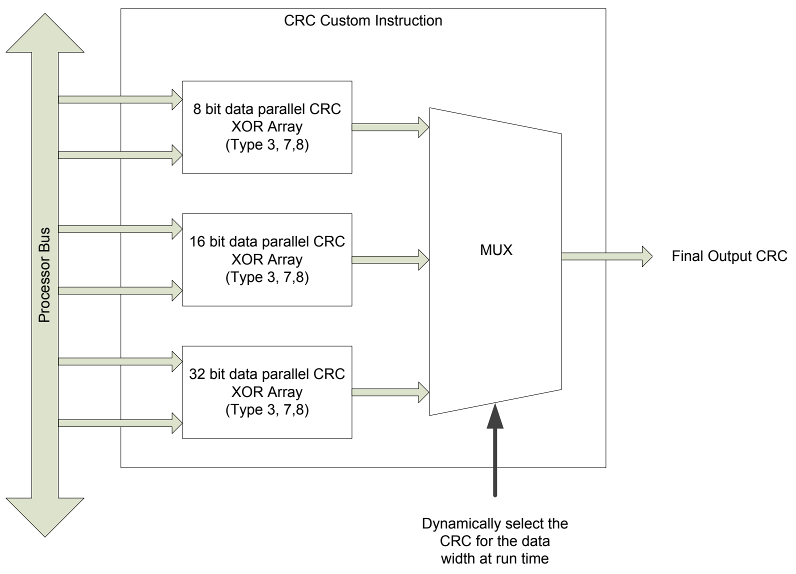

CRC Custom Instruction
Processor Bus
8 bit data parallel CRC XOR Array (Type 3, 7,8)
16 bit data parallel CRC XOR Array (Type 3, 7,8)
32 bit data parallel CRC XOR Array (Type 3, 7,8)
MUX
Final Output CRC
Dynamically select the CRC for the data width at run time

Figure 5.10: CRC hardware accelerator architecture

is that we can use software for implementation of a mixed-data width CRC selection algorithm, as discussed in section 5.9.1, and the parallel CRC generators for different data widths can be implemented in hardware as custom instructions.

The architecture consists of parallel CRC generators for 32 bit, 16 bit and 8 bit data which are selected dynamically by using an option called opcode extension in the NIOS II custom instruction set. The parallel CRC generators are designed using the equations as mentioned in section 4.1.1. Since the NIOS CPU is little endian, the CRC equations must include the necessary mappings to take care of the endianess. Necessary changes in software are also made to include the CRC custom instruction, which can be reconfigured for mixed data widths.

5.9.3 WLSB Hardware Accelerator

The LSB Hardware accelerator, which is a pure combinatorial data path, is implemented as a NIOS II custom instruction as proposed in the previous section. The LSB custom instruction calculates the number of bits needed to encode one field in one clock cycle. Other steps in the WLSB encoding, like finding minimum and maximum values from the window entries, are taken care of in the software. Checking various conditions and determining which equation to use, as discussed earlier in section 4.2, are also done in software.

5.10 Fast prototyping on an FPGA

The hardware-software solution is implemented in the form of a RoHC embedded system, as discussed in section 5.6. This embedded system is synthesized on an Altera II GX FPGA development board. Performance metrics, power dissipation, and design size are all captured from the synthesized RoHC. This board has the following components which are relevant to RoHC design:

1. Arria II GX EP2AGX260FF35 FPGA in the 1152-pin (FBGA) package with 244,188 LEs, 102, 600 Adaptive Logic Modules (ALMs) and 11,756 kbit on-die memory.
2. On-Board memory of 128 Mbyte 16-bit DDR3 memory, 1 Gbyte 64-bit DDR2, 2 Mbyte SSRAM, and 64 Mbyte flash memory.

5.11 HW-SW co-verification

The CRC and WLSB modules, which are identified as candidates for hardware design, are isolated and individual reference code is implemented, which will aid the hardware implementation. After hardware implementation these blocks are individually tested with a sample excitation and the functionality is verified.

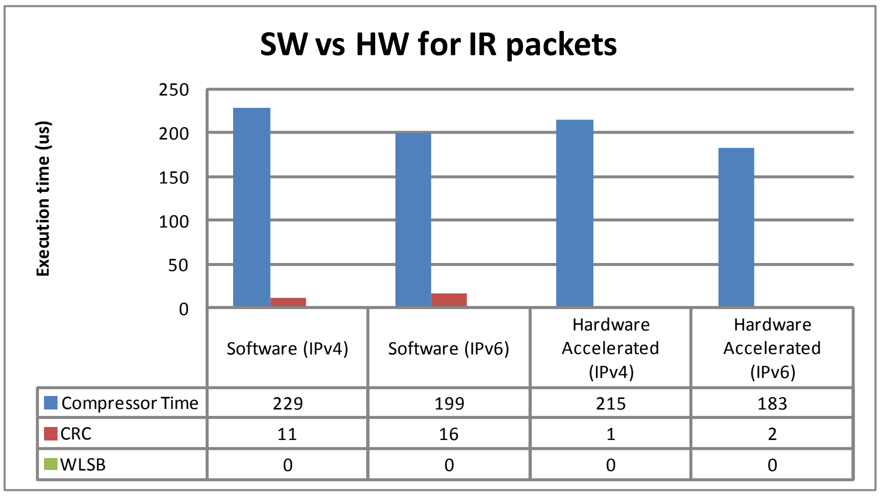
The SW-based RoHC design is first ported on FPGA and is verified with standard RoHC output vectors derived from the reference code, as mentioned in [10], using wireshark. Once the software design is verified the CRC and WLSB functionalities are removed and the respective hardware blocks are integrated with the software design as custom logic instructions in NIOS II-based embedded system. The RoHC functionality is cross-verified with the RoHC outputs of the reference code.

5.12 Results

RoHC is a performance-critical system. The HW-SW co-design methodology is used to do rapid prototyping of RoHC on an FPGA board. The performance is accelerated and evaluated with respect to throughput, capacity, memory usage, area, and power consumed.

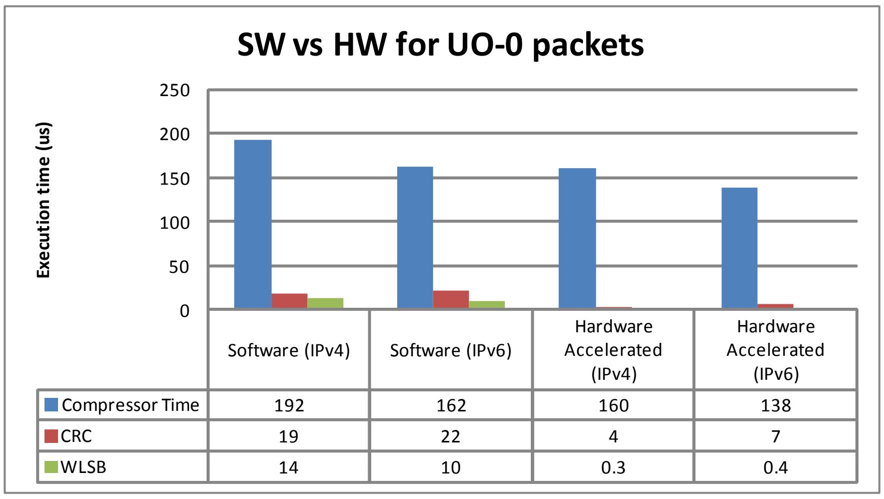
5.12.1 RoHC Throughput

The throughput is a measure of the average rate at which the RoHC system can process uncompressed headers and generate compressed RoHC header packets for a particular packet stream. It signifies how fast the RoHC can run and cater to one particular user.



	Software (IPv4)	Software (IPv6)	Hardware Accelerated (IPv4)	Hardware Accelerated (IPv6)
🟦 Compressor Time	229	199	215	183
🟥 CRC	11	16	1	2
🟩 WLSB	0	0	0	0

Figure 5.11: RoHC performance graph for IR packets



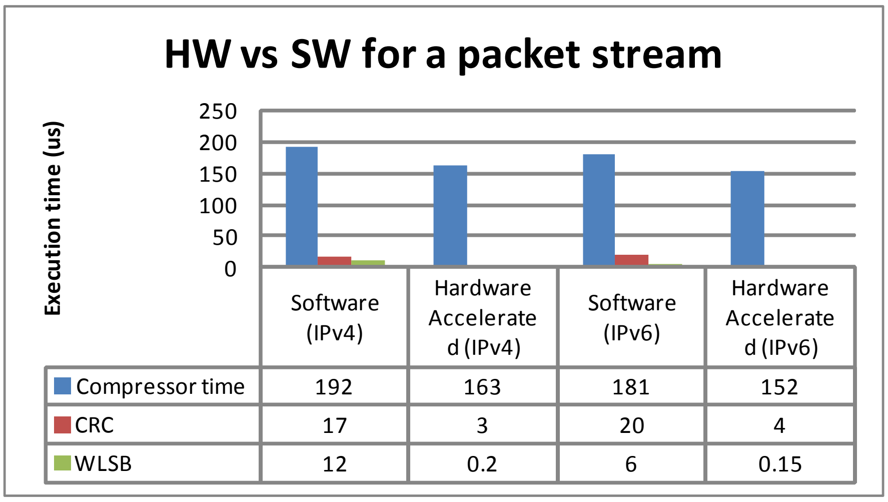
	Software (IPv4)	Software (IPv6)	Hardware Accelerated (IPv4)	Hardware Accelerated (IPv6)
🟦 Compressor Time	192	162	160	138
🟥 CRC	19	22	4	7
🟩 WLSB	14	10	0.3	0.4

Figure 5.12: RoHC performance graph for UO-0 packets

Here throughput is measured with respect to different RoHC packet types and also average increase in throughput across a communication stream of 10 packets.

The results for software vs. hardware-accelerated IR packets (the largest RoHC packet) and UO-0 packets (the smallest RoHC packet) are tabulated as shown in the plot in Figure 5.11 and Figure 5.12. We see that the performance increase, due to hardware acceleration, with respect to throughput ranges from 15-17% for IR packets to 20-22% for UO-O packets for both IPv4 and IPv6 streams.

We see a 15-20% average increase in throughput and a final throughput of $1/160\ \mu s =$ $6250\ packets/s$ is achieved for IPv4 and $1/150\ \mu s = 6666\ packets/s$ is achieved for



	Software (IPv4)	Hardware Accelerated (IPv4)	Software (IPv6)	Hardware Accelerated (IPv6)
🟦 Compressor time	192	163	181	152
🟥 CRC	17	3	20	4
🟩 WLSB	12	0.2	6	0.15

Figure 5.13: RoHC performance graph for a average packet stream

IPv6, as shown in the plot in Figure 5.13. This is for a scenario in which we consider a communication stream of 10 packets. It is applicable to both IPv4 and IPv6. The RoHC packets generated range from large IR packets to smaller UO-2, UO-1 packets, and the smallest UO-0 packets.

5.12.2 RoHC Capacity

Here RoHC capacity refers to the number of users that can be supported by a particular base station, or more precisely, a particular instantiation of the RoHC algorithm. RoHC capacity is an important metric for a base station since it has to service many users. RoHC capacity for VoIP traffic, in the sense of number of supported users, is calculated by the following formula:

$$\begin{aligned} Capacity &= \frac{\text{Frequency of VOIP frames}}{\text{Voice activity factor} \times \text{RoHC execution time}} \\ &= \frac{20ms}{0.5 \times \text{RoHC execution time}}. \end{aligned}$$

For a given user the frequency of arriving VoIP packets is around 50 packets/s so 20 ms is the duration of one VoIP packet. The voice activity factor is set to 50% because of silence suppression. Voice activity factor is the ratio of duration of talk spurts to silence duration and it signifies how many people talk in a session. It is typically set to 50% as both users cannot talk at the same time.

Considering the RoHC execution time of 160 μs to 200 μs as discussed before, the RoHC capacity will be 200-270 users.Thus, the capacity increases by 50-60 users when the RoHC software is hardware accelerated,when compared to pure software implementation. A base station running hardware-accelerated RoHC can support an increase in capacity of around 30% in this respect (this may differ based on the actual choice of implementation of RoHC libraries).

5.12.3 Memory

The amount of memory and memory bandwidth are critical performance factors in order for RoHC to support substantially higher amount of capacity. The RoHC module prototyping platform, the Altera Arria II GX, has support for both DDR2 and DDR3 external memories to address the problems of both the amount of memory required and access speed for high capacity.

The code footprint of the ported RoHC libraries with hardware accelerators, including the ten test VoIP packets is about 80-90 kB. This can be easily accommodated in on chip memory while external memory can be used to store the context memory for a large number of users. The external memory should support high bandwidth as the context data have to be stored and retrieved for many users from the external memory.

With the aim of supporting maximum capacity (which may be 1500 users supported by a single cell in an LTE system in the near future), with a RoHC context size of 360 bytes which is read and written into external memory and with he VoIP throughput which is 50 packets/s (20 ms to process one VoIP packet ignoring voice activity factor of 0.5) The memory bandwidth for the RoHC system for supporting maximum bandwidth is given by equation as shown below :

$$\begin{aligned} BW(RoHC) &= N \times context\ size \times 2(R/W) \times 8(byte) \times 50\ packets/sec\ (VoIP\ throughput) \\ &= N \times 360 \times 2 \times 8 \times 50 = 288kbit/sec \times N\ (Memory\ BW\ for\ 1\ user) \\ &= 1500 \times 288kbit/sec \approx 430Mbps\ (Memory\ BW\ for\ N = 1500\ users). \end{aligned}$$

With a DDR2 interface of 42.6 Gb/s @ 64-bit data bus and a DDR3 interface of 10.6 Gb/s @ 16 bit data bus, the memory bandwidth of the target device is more than enough to cater to the memory bandwidth needed to support maximum throughput.

5.12.4 Area

The RoHC embedded system is synthesized on an Altera II GX FPGA. The logic utilization in terms of ALUTs is as shown in Table 5.2. This includes the ALUTs for the

Table 5.2: Logic utilization of components of RoHC embedded system

Block	**Combinatorial ALUTs**	**Dedicated Logic Registers**	**Block Memory bits**
NIOS II CPU	2215	1680	325504
DDR 3 Memory controller	2805	2112	19072
JTAG UART	125	112	1024
On chip memory	36	2	1048576
Sys ID	26	17	0
Performance Counters	1071	711	0
WLSB Custom Instruction	257	0	0
CRC Custom Instruction	106	0	0
Misc	1178	792	0
Total	**7871 /205200 (4 %)**	**5433/205200(1 %)**	**1392064 /8755200 (16%)**

CRC and the WLSB modules. An ALUT consists of a 6-input LUT, an adder and a register pair, and is the basic cell of synthesis in the FPGA.

We observe that the number of ALUTs utilized by the entire RoHC system is about 4% of what is available on the board for combinational ALUTs and about 16% for block memory bits. In this small utilization by the RoHC system,the utilization of the hardware accelerators is even smaller. The HW accelerators consume roughly 15% of the ALUTs compared to the NIOS system (excluding peripherals). Thus, the addition of such accelerators (CRC, WLSB) will not change the FPGA utilization by much as they occupy negligible number of ALUTs compared to the entire system.

5.12.5 Power Consumption

Alteras PowerPlay Power Analyzer is used to estimate the power consumed by the device. This is important for thermal planning and power supply planning for the device.

For an average toggle rate set to 12.5% (default), power consumption readings are taken for the entire RoHC embedded system with NIOS II soft-core processor and all the peripherals including the custom instruction hardware accelerators like WLSB and CRC. These power readings are tabulated in the table 5.3. We see that the HW accelerators roughly consume 2% of the power compared to the entire system (excluding peripherals), which is negligible.

5.13 Observations

For hardware-software co-design of RoHC, with implementations of CRC and WLSB as hardware accelerators, the major bottleneck in the overall throughput of the system is the handling of uncompressed RoHC packets. These are the IR and IR-DYN packets. They only make use of the CRC HW accelerator and not WLSB HW accelerator. This is because, the fields are not WLSB encoded in IR and IR-DYN packets. However, in a

Table 5.3: Power consumption of components of RoHC embedded system

Block	**Total Thermal Power (mW)**
NIOS II CPU	115.84
DDR 3 Memory controller	223.67
JTAG UART	3.07
On chip memory	141.9
Sys ID	0.17
Performance Counters	10.26
Alt PLL	108.7
WLSB Custom Instruction	2.96
CRC Custom Instruction	0.99
Misc	51.51
Total	**659.07**

given traffic situation, the probability that the RoHC system has to send only IR packets for all the users at the same time is statistically very small. Therefore, the potentially critical impact of uncompressed RoHC packets on the overall throughput is averaged out.

The introduction of these hardware accelerators does not cost much in terms of area and power, as seen in section 5.12. Usually, the LTE physical layers is implemented in base stations with the help of a number of hardware accelerators, e.g, accelerators for the computation of CRCs, FFTs, etc. The addition of RoHC dedicated hardware accelerators could also be seriously considered as these would greatly boost the performance of RoHC.

The choice of going for hardware acceleration of RoHC is greatly influenced by the capability of the lower layers to handle the resulting capacity. If the scheduling in LTE is not capable of handling the traffic which the hardware-accelerated RoHC is capable of handling, then it is better to adjust the capacity (throughput) of the RoHC by using hardware accelerators for only a few functions. Around 20% overall increase in performance is obtained with the aid of CRC and WLSB hardware accelerators to the RoHC software. Even more increase in performance can be obtained if more components are partitioned to be put in hardware which is further investigated in next chapter.

Chapter 6

Full-HW Solution

6.1 Introduction

In this chapter, the implementation of the RoHC algorithm for the compressor in full hardware is discussed. An efficient hardware architecture that can cope with the throughput requirement of an LTE base station is proposed. The implementation covers most of the encoding methods defined in RFC3095, [1], for profile 1. However, the main functionality of profile 0 and 2, explained earlier in section 3.2, are also implemented but their details are omitted in this context. Table 6.1 below summarizes the coverage of the current implementation.

The rest of this chapter is organized into four sections. In section 6.2, implementation-specific parameters of the RoHC algorithm are discussed. In section 6.3, the top-level architecture of the full hardware solution is presented and then explained in detail. In section 6.4, a more powerful hardware architecture is presented. In section 6.5, key results and findings of the full-HW solution are presented and discussed.

Table 6.1: Full-HW implementation features

Feature	**Status**
Mode	U, O and R
Profile	1
Feedback processing	Not supported
IP level	2
IPv4	Supported
IPv6	Supported
Compression list	Not supported

6.2 RoHC Implementation-Specific Parameters

As mentioned earlier in chapter 3, some parameters defined in the standard, [1], such as timeout counters, sliding-window size, and the optimistic approach are implementation-dependent features. Varying these parameters while preserving the channel conditions influences the overall performance of the algorithm. Table 6.2 shows these parameters and their chosen values in this implementation.

Although choosing the optimum values for these parameters is not part of this study, an outline of why these values are chosen is given. The sliding-window size is probably the most crucial parameter as it affects the compression ratio. The compression ratio is of course the reason this algorithm was developed in the first place. This is because the sliding-window size influences the selection of which compressed packet to be sent. For instance, by choosing the sliding-window size to be 16, it might not be possible to send UO-0 packets since the required number of bits to encode the sequence number with respect to 16 values is very likely to be more than 4 bits. The UO-0 packet is the smallest packet RoHC can send and it can only transmit 4 bits of the sequence number.

Since the transmitted data in LTE downlink are protected by strong checksums and with the support of retransmission mechanisms, there is a rather small risk that the decompressor in the LTE uplink has lost a packet. Thus, a sliding-window size of 4 elements seems reasonable for both boosting the compression ratio and reducing the processing and HW resources. Using the same argument, the other parameter values are chosen by relying on the efficiency of the LTE system, which is the host of this implementation, in protecting the transmitted packets.

Table 6.2: RoHC implementation parameters

Parameter	**Value**
IR timeout counter	64
FO timeout counter	32
Optimistic approach	2
Sliding-window size	4

6.3 Top-level Architecture of the One-Stage Full-HW Implementation

The full hardware architecture of the RoHC algorithm is divided into a controller stack and data-path accelerators connected through a shared bus or point-to-point connection as shown in Figure 6.1. The controller stack is nothing but a set of functions connected together in a sequential fashion as shown in the figure.

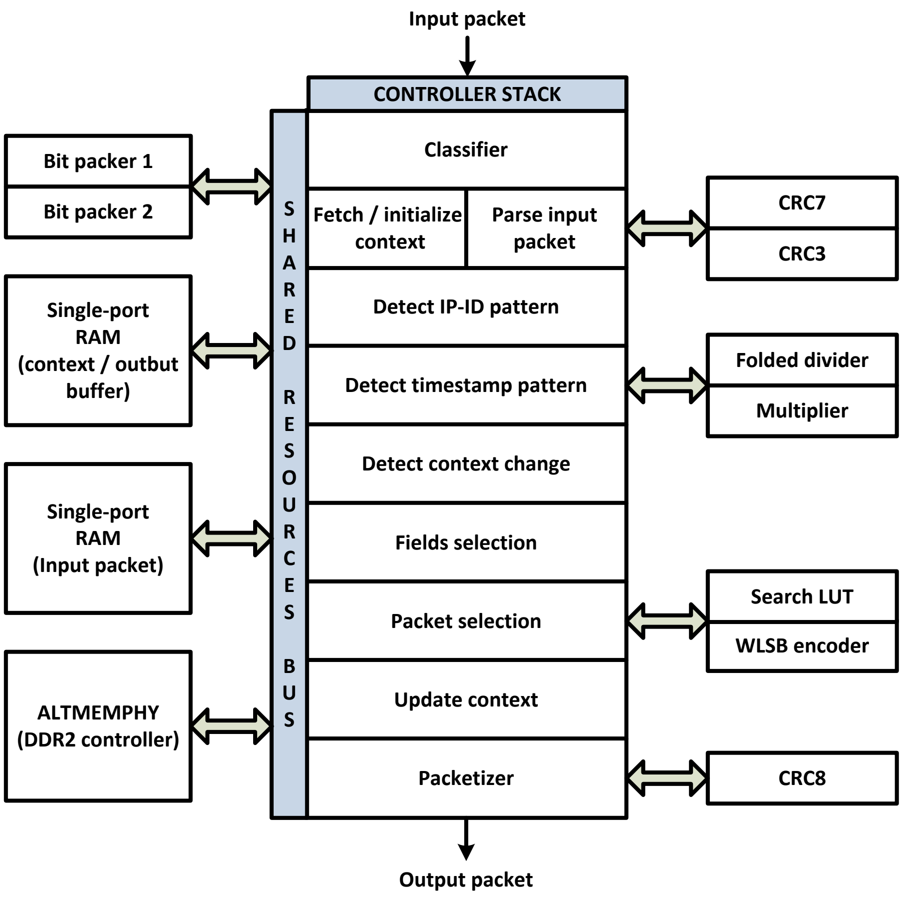


Figure 6.1: Top-level architecture of the full-HW solution

Since most of the operations in the RoHC algorithm are based on comparing the context data and the received packet data; in order to accelerate these operations these data are stored in separate RAMs. However, the context RAM might also be used as a temporary storage for the output packet. Thus, only two single-port RAMs are used in this implementation as shown in Figure 6.1.

For fetching and updating the context data from and to the external DDR2 memory, an Altera DDR2 controller Intellectual Property (IP), provided in Quartus II software, is used. This DDR2 controller is referred to as ALTMEMPHY in Figure 6.1. The compressor can be interfaced to the outside world using the Altera standard memory-mapped interface, [11].

6.3.1 Classifier

This classifier is responsible for assigning a profile to the input packet by inspecting the header fields of the packet. Also, the classifier outputs the User ID (UID), the Stream ID, the packet length, the payload position, the identified protocol, and their offsets in the packet to the next controller in the stack. The Stream-ID is the Synchronization Source Identifier (SSRC) field in the RTP header when profile 1 is assigned.

The classifier assigns profile 1 to the packet if an RTP/UDP/IP protocol stack is identified, whereas profile 2 is assigned to the packet if an UDP/IP protocol stack is identified. Profile 0 is assigned if none of the previous protocol stacks is found or the IP protocol does not meet the RoHC specification for compression.

Classification of IPv4

An IPv4 protocol is identified from the *version* field in the first octet of the IPv4 header. It can also be identified from the *next header* or the *protocol* field in the outer IPv4 or IPv6, respectively, if two IP levels[1] are found in the packet. The identification of the protocol fails if the number of octets in the received packet cannot cover all the header information for a typical IPv4 header size.

If an IPv4 protocol is identified, profile 0 is assigned to the packet unless the *fragmentation offset* and the *flags* fields, excluding the *don't fragment* field, are not zero, the *protocol* field is a UDP protocol, and the *IHL* field is 5.

Classification of IPv6

An IPv6 protocol is classified from the *version* field in the first octet of the IPv6 header. It can also be identified from the *next header* or the *protocol* field in the outer IPv4 or

[1]The inner and outer IPs are used to connect users from two private networks together through a so-called IP tunnel.

IPv6, respectively, if two IP levels are found in the packet. The identification of the protocol fails, if the the number of octets in the received packet cannot cover all the header information for a typical IPv6 header size.

If an IPv6 protocol is identified, profile 0 is assigned to the packet unless the *next header* field is a UDP protocol.

Classification of UDP

A UDP protocol is classified from next header or the protocol field in the inner IPv4 or IPv6 respectively. The identification of the protocol fails, if the number of octets in the received packet cannot cover all the header information for a typical UDP header size.

Profile 0 is assigned if the identification of the UDP protocol fails.

Classification of RTP

An RTP flow is identified from the *version* field in the first octet of the RTP header which must be 2. The identification of the protocol fails, if the the second octet in the header, M and payload type fields, is equal to 200 or 201 as it conflicts with the Sender Report (SR) or Receiver Report (RR) payload type for the RTP Control Protocol (RTCP). The identification also fails, if the number of octets in the received packet cannot cover all the header information for a typical RTP header size.

If an RTP flow is identified, profile 2 is assigned to the packet unless the *counter* field is equal to zero. Otherwise, Profile 1 is assigned to the flow. The *counter* field refers to the number of the RTP Contributing Source Identifiers (CSRC) found in the packet. Hence, the compression list of an RTP-CSRC field is not supported in this implementation.

6.3.2 Fetching or Initializing The Context

Each user has a context stored in the external memory which is fetched when a packet from the same stream is received. The assigned profile and the UID are used to calculate the physical address of the context and its size in the DDR2 memory. This controller is also responsible for initializing the context if the packet is the first packet in the stream. The packet is considered the first packet in the stream, if the protocol stack and the Stream-ID, which were identified earlier by the classifier, do not match the ones stored in the external memory. Figure 6.2 shows the structure of the profile 1 context which consists of 81 words. A word is considered 32 bits in this implementation.

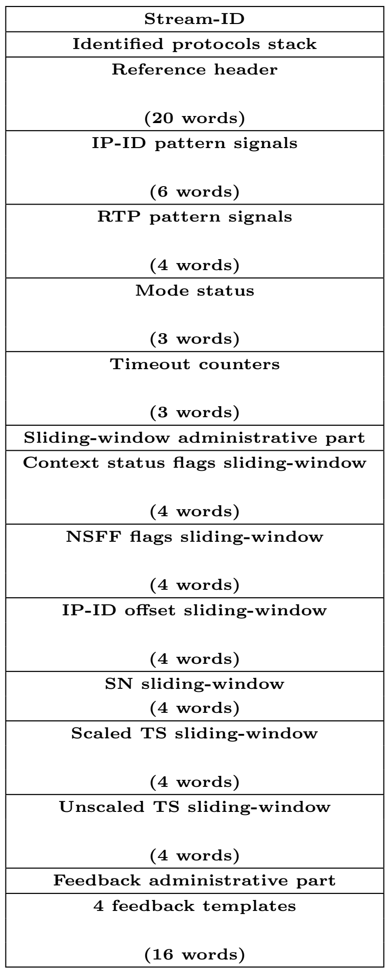


Figure 6.2: Context structure for profile 1

When the context is initialized, some fields need to be set to a specific value. Table 6.3 lists these fields and their initial values in a newly created context. The IP-ID pattern fields and the RTP pattern fields are discussed in details in subsection 6.3.4 and 6.3.5, respectively.

6.3.3 Parsing The Input Packet

This controller works in parallel with the previous controller in subsection 6.3.2 as they are data independent. It is responsible for parsing the input packet into static-dynamic fields (*parsing operation 1*) and extracting the fields required to calculate CRC3 and CRC7 (*parsing operation 2*). Since both operations share a significant number of fields, a scheduling to run both operations in parallel is done to reduce the number of times the memory is accessed, which in return reduces both the processing time and the

	Field	Initial Value
IP-ID pattern fields	NBO_0	1
	RND_0	0
	NBO_1	1
	RND_1	0
RTP pattern fields	TS_STRIDE	1
	TS_JUMP	1
Mode status fields	Mode	Unidirectional
	Mode transition	Done

Table 6.3: Initialization of context fields

power.

In order to reduce the memory resources for *parsing operation 1*, fields of the same category, i.e., static or dynamic, are concatenated together in a 32 bits memory-word. Similarly, the extracted fields in *parsing operation 2* must be concatenated together without any gaps or padding before the CRC calculations. Both parsing operations require bit masking, shifting and concatenating operations. These three logical operations are executed in one clock cycle using the hardware presented in Figure 6.3, which would be referred to as the bit-packing hardware throughout this work. The bit-packing hardware is used in other controllers in the stack. Hence, it is among the shared resources in Figure 6.1.

The bit-packing hardware has four main control signals *msb*, *lsb*, *shift* and *sel_lft_over*. The *msb* and *lsb* signals are used to generate a mask for the bits of interest in the input data. Then, the masked input data is shifted right or left by the barrel shifter and concatenated together with what was previously stored in the register. The result of concatenation might be taken out directly in the same clock cycle or stored in the register. If the register does not have enough space to accommodate all the input bits, it is possible to save the left-over bits in the concatenation and take the results of the concatenation out in the same clock cycle.

6.3.4 Detecting IP-ID Pattern

The IP-ID is one of the IPv4 fields that can be compressed by communicating its difference relative to the sequence number, as discussed earlier in subsection 3.7.4, using the WLSB encoding method. However, The IP-ID might not be compressed if it is randomly generated in the network layer. In addition, the IP-ID must be converted to the network byte order before IP-ID offset encoding is done, if it was transmitted in little-endian order from the network layer. Network byte order is also referred to as big-endian order in non-networking fields.

To detect the IP-ID pattern, the previously detected NBO and RND flags are stored

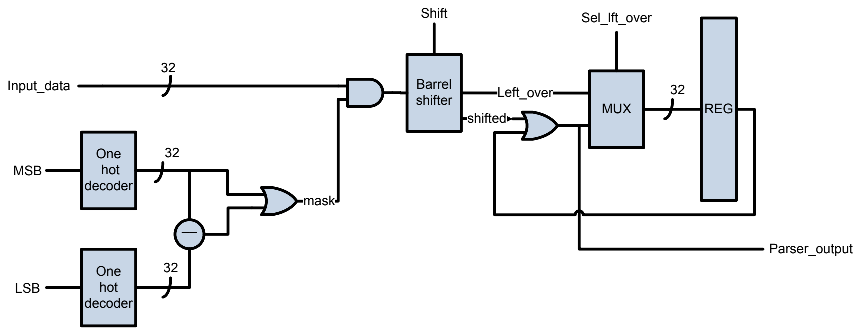


Figure 6.3: Bit-packing hardware

in the context. Moreover, two references to the IP-ID and the SN from the last two packets are stored in the IP-ID pattern part of the context. Below are some notations used throughout this work which are consistent with the notations used in the RFC3095 [1].

context(field): represents the value of the field stored in the context.

hdr(field): represents the value of the field with respect to the input packet.

The following procedure is used to detect the IP-ID pattern:

1. Calculate *offset*, *ref_offset*$_1$ and *ref_offset*$_2$ with respect to *context(nbo)*.
2. If any of the offsets calculated in 1 matches any of the other two offsets, the value of the *context(nbo)* and 0 are assigned to *hdr(nbo)* and *hdr(rnd)*respectively.
3. Otherwise, calculate *offset*, *ref_offset*$_1$ and *ref_offset*$_2$ with respect to the inverted value of the *context(nbo)*.
4. If any of the offsets calculated in 1 matches any of the other two offsets, the inverted value of the *context(nbo)* and 0 are assigned to *hdr(nbo)* and *hdr(rnd)* respectively.
5. Otherwise, both *hdr(nbo)* and *hdr(rnd)* are set.

Both *ref_offset*$_1$ and *ref_offset*$_2$ are calculated from the two IP-ID pattern references stored in the context as in equation 6.1 and 6.2, respectively.

$$ref\_offset_1 = swap(context(ip - id_1), nbo) - context(sn_1), \tag{6.1}$$

and

$$ref\_offset_2 = swap(context(ip - id_2), nbo) - context(sn_2). \tag{6.2}$$

The *offset* is calculated from the input packet header as in equation 6.3.

$$offset = swap(hdr(ip - id), nbo) - hdr(sn). \tag{6.3}$$

The swap function is used to change the byte order of the IP-ID and is defined as in equation 6.4 below:

$$swap(x, nbo) = \begin{cases} \sum_{i=0}^{7} 2^{i+8} \cdot x(i) + \sum_{i=8}^{15} 2^{i-8} \cdot x(i) & \text{if } nbo = 0 \\ x & \text{otherwise} \end{cases} . \tag{6.4}$$

All of the following, *hdr(nbo)*, *hdr(rnd)* and *offset*, are stored in the packet RAM.

6.3.5 Detecting RTP Pattern

The timestamp of an RTP packet behaves differently depending upon the application that the RTP is used for, e.g. audio or video application. For instance, when the RTP protocol is used with an audio application, the timestamp is increased by the number of samples which the payload carries regardless of whether the packet is transmitted or dropped. See the case of silence suppression, as explained in [12]. If the RTP protocol is used in conjunction with a video application, the timestamp doesn't change in the packets that carries the same video frame. However, the sequence number is incremented by one if and only if the packet is sent. Also, RTP packets might be sent in a different order than the order they were sampled, e.g., in the case of MPEG encoding is used. Figure 6.4 shows three examples of the timestamp behaviour in video and audio applications.

Another special case handled here is when a retransmission of an RTP packet is suspected. The RTP receiver may request for a packet retransmission from the RTP sender. As defined in [13], the RTP sender retransmits the same original sequence number as an indication of a packet retransmission.

To detect the RTP pattern, several flags and fields are used to track the behaviour of the timestamp such as *ts_stride*, *ts_scaled*, *ts_offset* and *ts_jump*. The notations of *ts_stride*, *ts_scaled*, and *ts_offset* are explained earlier in subsection 3.7.2. *Ts_jump* is used to refer to the existence of a silence-period in an audio stream, a new video frame has started, or the timestamp doesn̂increase by an integral multiple of the ref_ts_stride.

The expected pattern of *hdr(ts)* is when it is increased by *context(ts_stride)* when it is compared to *context(ts)*, whereas, *hdr(sn)* is expected to increase by one when it is compared to *context(sn)*. Providing that the former conditions are true, *hdr(ts_stride)* is assigned the value of *context(ts_stride)*. Otherwise *hdr(ts_stride)* is calculated as in equation 6.5 below:

$$hdr(ts\_stride) = \begin{cases} ts\_delta - sn\_delta \times context(ts\_stride) & \text{if } ts\_jump = 1 \\ context(ts\_stride) & \text{otherwise} \end{cases} . \tag{6.5}$$

ts_delta and *sn_delta* are calculated in equation 6.6 and 6.7 below:

$$ts\_delta = \begin{cases} hdr(ts) - ref(ts) & \text{if packets are in order} \\ 2^{32} - (hdr(ts) - ref(ts)) & \text{otherwise} \end{cases} , \tag{6.6}$$

and

$$sn\_delta = \begin{cases} hdr(sn) - ref(sn) - 1 & \text{if packets are in order} \\ (ref(sn) - hdr(sn) - 1) & \text{otherwise} \end{cases} . \tag{6.7}$$

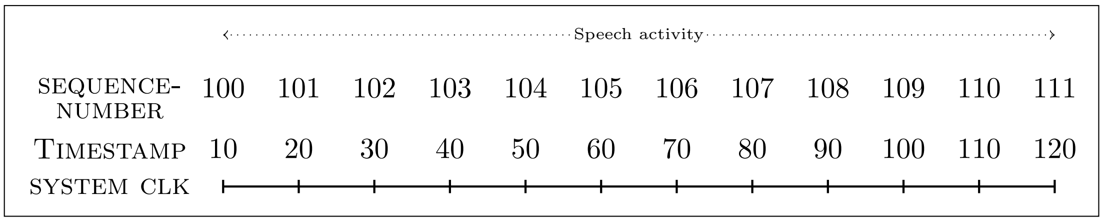
In case of a video frame transmitted in multiple packets, the *ts_stride* may equal zero. In such a case, *ts_scaled* is always the unscaled timestamp. Equation 6.8 below then takes precedence over equation 3.3 in subsection 3.7.2:

$$ts\_scaled = \begin{cases} hdr(ts) \div hrd(ts\_stride) & \text{if } hdr(ts\_stride) \neq 0 \\ hdr(ts) & \text{otherwise} \end{cases} . \tag{6.8}$$

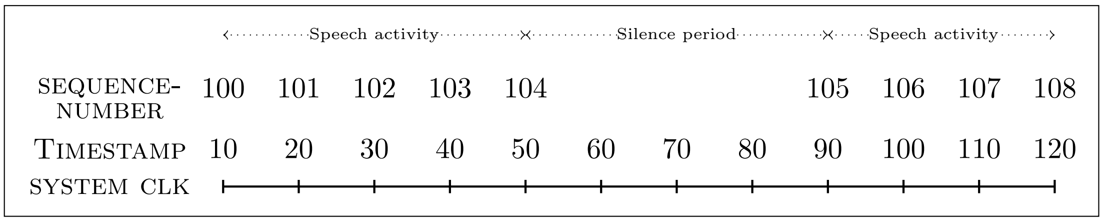
Another field that is important in tracking the timestamp behaviour is the *ts_offset*. The *hdr(ts_offset)* is calculated as follows:

$$hdr(ts\_offset) = \begin{cases} hdr(ts) & \text{if } hdr(ts\_stride) = 0 \\ context(ts\_offset) & \text{otherwise if } hdr(sn) \neq context(sn) \\ hdr(ts) \bmod hrd(ts\_stride) & \text{otherwise} \end{cases} . \tag{6.9}$$

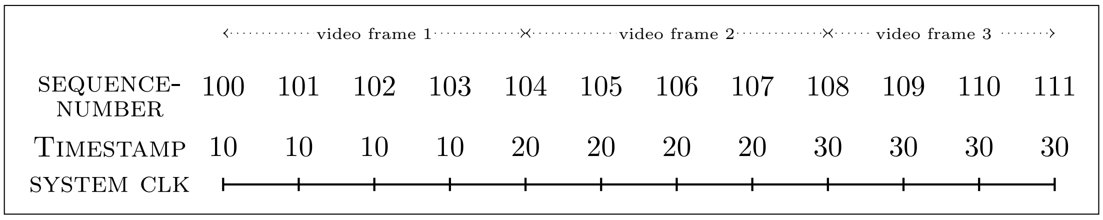
All of the following *hdr(ts_stride)*, *hdr(ts_offset)* and *hdr(ts_jump)* are stored in the packet RAM. Regarding the division and modulus operations needed in equation 6.8 and 6.9 respectively, a time multiplexed cordic integer divider is used for this purpose as shown in Figure 6.1.


Speech activity
SEQUENCE-NUMBER 100 101 102 103 104 105 106 107 108 109 110 111
TIMESTAMP 10 20 30 40 50 60 70 80 90 100 110 120
SYSTEM CLK

(a) Continues voice frames


Speech activity
Silence period
Speech activity
SEQUENCE-NUMBER 100 101 102 103 104 105 106 107 108
TIMESTAMP 10 20 30 40 50 60 70 80 90 100 110 120
SYSTEM CLK

(b) Voice frames with silence period


video frame 1
video frame 2
video frame 3
SEQUENCE-NUMBER 100 101 102 103 104 105 106 107 108 109 110 111
TIMESTAMP 10 10 10 10 20 20 20 20 30 30 30 30
SYSTEM CLK

(c) Video frames

Figure 6.4: TS-SN pattern in video and voice frames

6.3.6 Detecting Context Change

To detect the context change, different fields from the context are checked with their equivalent in the received packet. The results of this extensive comparison process are stored in flags; these flags are then used by other controllers in the stack.

Two sets of the same flags are created, Sliding-Window (SW) flags and Field Selection (FS) flags. The SW flags are later stored in the sliding-window when the context is updated. On the other hand, the FS flags are only stored temporarily since they are only used in the process of packet and fields selection.

The values of these flags before this operation come from the feedback processing. However, since the feedback processing is not implemented, the values of these flags are initialized to zero for all the packets in the stream except for the first one. Also, the values of both SW flags and FS flags are equal after this operation. Nevertheless, the values of FS flags might be altered in the fields selection process, as it is explained in the next subsection.

The different flags are defined below:

1. Context status flags:

   Stat: This flag is set, if the packet is the first packet in the stream or the static part of the reference header in the context does not match the received packet's static part.

   Dyn: This flag is set, if the packet is the first packet in the stream

   Mt: This flags is set, if the mode transition is pending or initiated.

   Nack: This flags is not changed here.

2. IP flags:

   Nsff_tos/tc: This flag is set, if the *hdr(tos)* in IPv4 or the *hdr(tc)* in IPv6 does not equal *context(ttl)* or *context(tc)*.

   Nsff_ttl/hl: This flag is set, if the *hdr(ttl)* in IPv4 or the *hdr(hl)* in IPv6 does not equal *context(ttl)* or *context(hl)* respectively.

   Nsff_df: This flag is set, if the *hdr(df)* in IPv4 does not equal the *context(df)*.

   Nsff_nbo: This flag is set, if *hdr(nbo)* does not equal *context(nbo)*.

   Nsff_rnd: This flag is set if *hdr(rnd)* does not equal *context(rnd)*.

   To support two IP-levels, two sets of the IP flags are created, i.e., one for the inner IP and the second for the outer IP.

3. UDP flags:

Nsff_checksum: This flag is set, if the *hdr(checksum)* does not equal *context(checksum)* and none of them equals to zero.

4. RTP flags:

   Nsff_p: This flag is set, if the *hdr(p)* does not equal *context(p)*.

   Nsff_x: This flag is set, if the *hdr(x)* does not equal *context(x)*.

   Nsff_pt: This flag is set, if the *hdr(pt)* field does not equal *context(pt)* field and non of them equal zero.

   Nsff_tss: This flag is set, if *hdr(ts_jump)* is equal to zero and *context(ts_stride)* does not equal *hdr(ts_stride)*.

   Nsff_offset: This flag is set, if the *hdr(ts_stride)* or *hdr(ts_offset* (or both) do not equal *context(ts_stride)* or *context(ts_offset)*, respectively.

   If this flag is set, the unscaled TS is encoded using the WLSB encoder. This flag masks the effect of the nsff_ts_wlsb flag below.

   Nsff_ts_wlsb: This flag is set, if the *hdr(ts_jump)* is equal to one. When this flag is set, the scaled TS is encoded using the wlsb encoder.

To accumulate the results of previous comparisons, a logical OR operation and a register are needed. Since these two elements already exist in the bit-packing hardware, it is used here as well.

6.3.7 Fields Selection

As discussed earlier in section 3.5, the compressor does not transit to a forward state until it is fairly confident that the decompressor context is in synchronization with its context. This forwards transition is analogue to the number of times the same field (or fields) are transmitted before the compressor assumes that it is safe to stop transmitting a specific field (or fields). This confidence is achieved through different means depending on the operational mode. In this controller, the fields which need to be transmitted to the decompressor are selected with the help of the flags stored in the context and the flags resulted from the controller in subsection 6.3.6.

In case of the unidirectional or bi-directional optimistic mode, the FS flags are merged[2] with the *context(sw_flags)* from previous packets. This merging operation is analogue to the so called optimistic approach. Since the optimistic approach in this implementation is only 2, the FS flags are merged with *context(sw_flags)* of the last two context-updating packets. As explained earlier, the optimistic approach refers to the number of time that a specific field is sent to the decompressor. Note that in this case only the FS flags are

[2]Logical OR operation is applied on the flags need to be merged.

changed; the SW flags are not altered by this controller. The SW flags are stored in the sliding-window part when the context is updated.

In case of the reliable mode, the SW flags are merged with *context(sw_flags)* of the last context-updating packet. This is to keep sending the same field until an acknowledging feedback is received and the acknowledged elements in the window are deleted. The operation of deleting the acknowledged elements in the sliding-window is done in the feedback processing, in case it were to be implemented. However, the FS flags are assigned the value of the SW flags after the merging with the old flags is done. The SW flags are stored later in the sliding-window part of the context in case the selected packet is a context-updating packet.

After merging the flags, the following operations are done on the FS flags:

1. If the *Nack* bit is set among the FS flags, the *tss* bit flag is set to send *ts_stride* in UOR-2-X-EXT3, IR or IR-DYN packets.
2. If the *checksum* bit is set among the FS flags, the *dyn* bit is set to send IR-DYN packet.

6.3.8 Packet Selection

The process of selecting the right packet to send is quite important to ensure a continuing successful compression and decompression. As discussed earlier for profile 1, there are 38 different compressed packets with different characteristics that the RoHC compressor can select from.

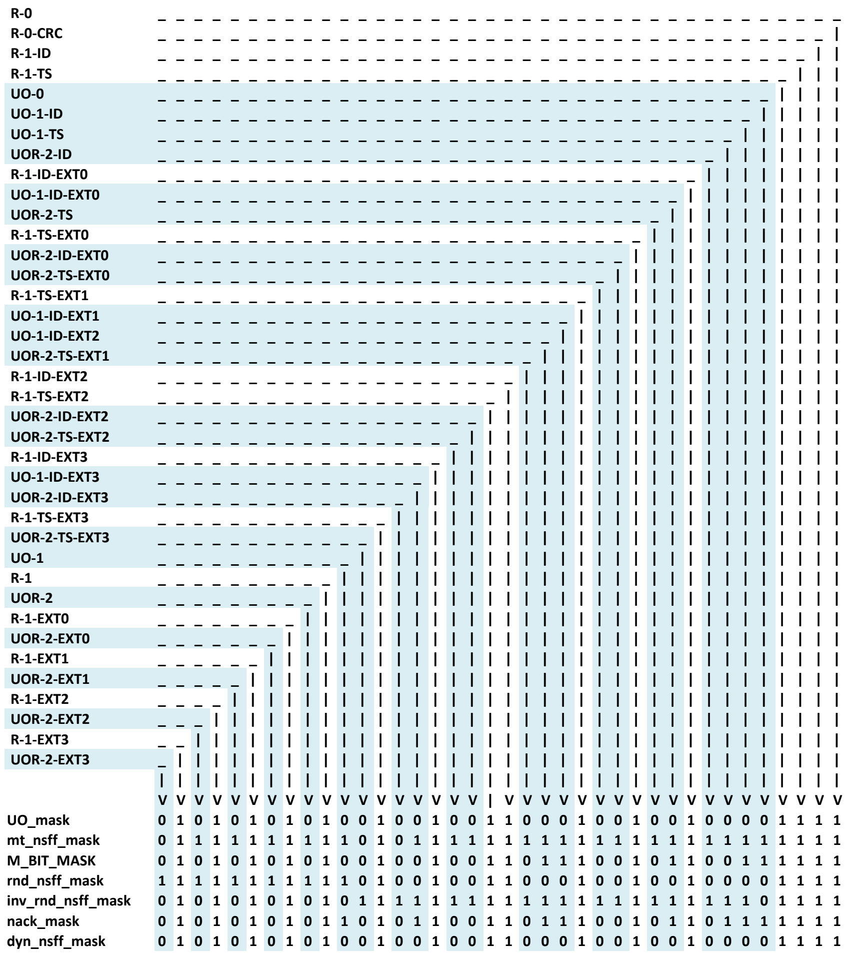
Unless an initialization packet must be sent, packet selection of a compressed packet passes through three main processes. Initialization packets are sent depending on the *stat* and *dyn* bits in the FS flags and the time-out counters in case of a unidirectional transmission. The explanation of the three main processes is as follows:

Excluding unsuitable packets

Not all the packets are suitable for the selection process. This depends on the operational mode, the status of mode transition, and which fields are selected for transmission.

One 38-bits register, referred to as the *disabled_pkt* register, is initialized to zero before entering this process. Each bit in the register refers to one compressed packet type. When a packet is excluded in the register, its bit is set to one. Figure 6.5 shows seven masks that might be applied to the register to exclude unsuitable packet types from the selection process when the operational mode is the unidirectional or the bi-directional optimistic mode. These masks are applied provided that the conditions in their prefixes are true. For instance, if the RTP-M bit in the received packet is set, the M_bit_mask from Figure 6.5 is applied to the *disabled_pkt* register to exclude all the packets that cannot transmit the RTP-M bit.

Since the reliable mode uses some different packet types than the other two modes, masks values are different than the ones in Figure 6.5.



```
UO_mask            0 1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1
mt_nsff_mask       0 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
M_BIT_MASK         0 1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 1 1 1 1 1
rnd_nsff_mask      1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1
inv_rnd_nsff_mask  0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
nack_mask          0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 1 1 1 1
dyn_nsff_mask      0 1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1
```

Figure 6.5: Unidirectional and bi-directional optimistic mode packets' masks

Minimal bit encoding using WLSB

SN, IP-ID, and TS are the fields which may be encoded using the WLSB encoder. However, some special cases must be checked for IP-ID and TS as below:

1. For any existing IPv4 header in the protocol stack whose IP-ID is detected to be random, the IP-ID field is sent as-is without any encoding.

2. The timestamp might be compressed by encoding either the *ts_scaled* or the original timestamp using the WLSB encoding method. The *nsff_offset* and *nsff_wlsb* bits in the FS flags control this operation.

The WLSB encoding method consists of the following operations which run in a pipeline:

1. Finding the minimum and the maximum reference values stored in the sliding-window for a specific field. This operation might take at most 4 clock cycles if 4 references exist in the sliding-window.
2. Finding the maximum number of bits required to encode a given field with respect to both minimum and maximum reference values found in previous step. This operation takes at most two clock cycles if the minimum and the maximum values are not equal. The LSB encoder in Figure 4.2 is used in this step.

   The process of searching the bit vector of the *disabled_pkt* register starts from least significant bit since the most desirable packet which have high compression ratio are ordered from the right to the left.

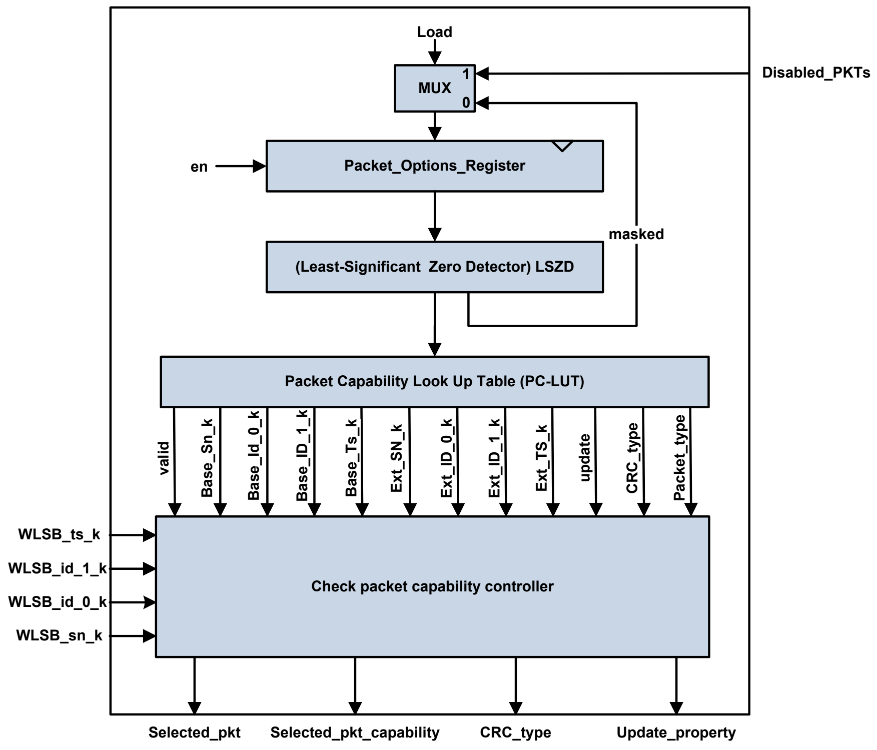


Figure 6.6: Packet capability check

Selecting the best packet

In the previous two processes, all the packets that are unsuitable for selection are disabled and the number of bits required to encode all of the SN, the IP_ID and the TS is calculated. In this process, the best packet which can transmit the number of bits calculated for the selected fields is chosen from the allowable packets in the *disabled_pkt* register.

Selecting the best packet requires checking if the capabilities of the packet in question meet the requirements. This process is repeated until a packet that meets the requirements is found.

To find a suitable packet in the register, a Least Significant Zero Detector (LSZD) is used. After finding a non-disabled packet in the *disabled_pkt* register, the corresponding bit is disabled. If the packet capabilities extracted from a look-up table meet the requirements, the packet is considered. If none of the compressed packet types meets the requirements, an IR-DYN packet is selected for the transmission. Fig 6.6 shows a block diagram of the hardware needed for this process. The output of the LSZD is one-hot coded; therefore, there is no need for a decoder in the PC-LUT.

6.3.9 Update Context

Different memory sections are updated depending on whether the selected packet is a context-updating packet or not. Provided that the selected packet has the updating property, almost all the sections of the context memory are updated. On the other hand, only the IP-ID pattern section is updated if the packet is not a context-updating packet. The updated sections are directly communicated to the DDR2 memory through the interfacing memory controller the ALTMEMPHY.

The updating of the sliding-window part works the same way the First-In-First-Out (FIFO) memory works. For instance, if the sequence number field is added, the *counter* and the *pointer* in the administrative part of the sliding-window is altered to reflect the number of elements and the position of the last element in the sliding-window. Thus, there is no need to rewrite all the pre-existing elements to the sliding-window or change their positions in the DDR2 memory.

6.3.10 Packetizer

This is the last controller in the stack. Here, the packet selected by the previous controller is packetized. Fig. 6.7 shows a flow chart of the packetizer operations. Two intersecting flows can be recognized in the flow chart, i.e., one for the initialization packets and one for the compressed packets.

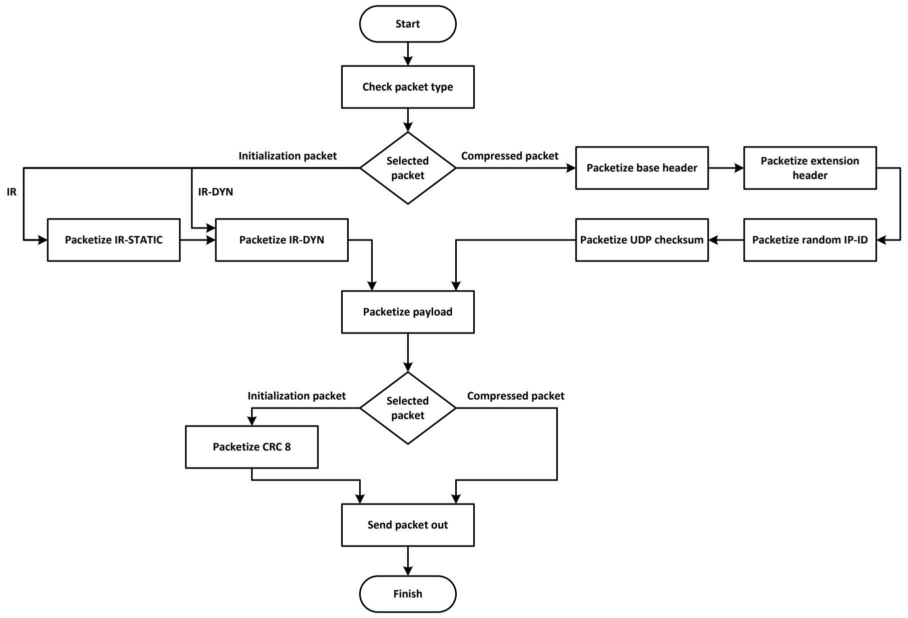


Figure 6.7: Packetizer flow chart

For the packetization process of the initialization packets, a CRC8 needs to be calculated over the entire packetized header (not the payload) before adding it to its place holder in the third octet of the packetized header. Therefore, in order to add the missing CRC8 field, the packetized packet is temporarily buffered in the output memory before sending it out. The calculation of CRC8 runs in parallel with the packetization process of the static and dynamic fields.

The packetization process needs the same logical operations used in the header parsing. Thus, the bit-packing hardware from the shared resources is used here as well for stuffing variable-length bit fields together.

6.4 Four-Stages Pipeline Architecture

In [3], the base station link speed is expected to reach 2.5 Gbps in the near future. This requires the RoHC compressor to handle an input rate of 2.6 million packets per second, assuming a packet size of a 120 bytes. The full-HW design presented in the previous section cannot handle such a high-speed link and a more powerful design is needed. Figure 6.8 shows a four-stage pipeline architecture that can compress at least three time the number of packets that the one stage hardware design can compress.

The pipeline architecture can process four packets at the same time. The pipelining is achieved by dividing the controller stack into four stages and removing the data

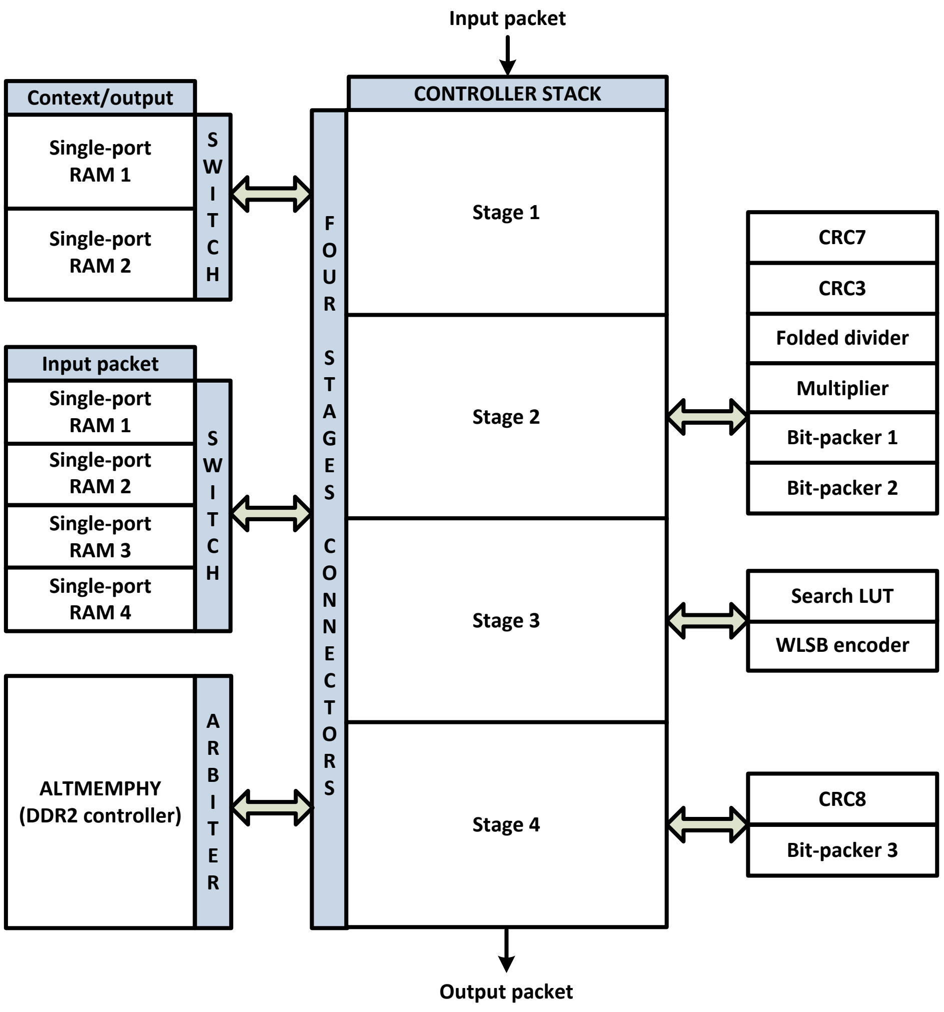


Figure 6.8: Four stage pipeline architecture

dependency between the stages. This requires extra storage and hardware resources. Since each stage in the controller stack needs the data of the input packet under its processing, four single-port RAMs are connected to the four stage controller through a full-crossbar switch to provide simultaneous transmissions. Similarly, a full-crossbar switch is needed to connect the context data to stage 2 and stage 3 in the controller stack. Also, an arbitration on the external memory controller is needed assuming that the context is fetched and updated in two different stages.

6.5 Results

The proposed one-stage full-HW design in 6.3 for the RoHC compressor is described in structural VHDL code and synthesised using a Quartus II compiler. The functionality of the design is verified on ARRIA II GX board by interfacing the design to a NIOS II micro-controller which runs the test bench written in C language. The interface between the NIOS II and the design follows the avalon memory-mapped interfacing protocol explained earlier in section 5.7. [3]

Table 6.4: One-stage Full-HW implementation results of RoHC compressor

Parameter	**Value**
Throughput[packet/second]	244k
#ALUT	7477
Internal memory usage [KB]	1
# Flip-Flop	2533
DDR2 bandwidth usage	DDR2
Memory bandwidth[kbps]	212.8
Power[Watt]	0.9 @ 90MHz
Max frequency[MHz]	100

Table 6.4 shows that the maximum operational frequency of the RoHC compressor is 100 MH_z and the total ALUT used is 7477. The design can compress up to 244 kilo-packet per second. The analysis of the key results are shown in Figure 6.10. The following subsections discuss the key results in more details.

6.5.1 Throughput and Capacity

The throughput is calculated in term of the number of packets that the design can compress in one second. Table 6.4 shows that the execution time of one packet might take at most 4.1us when an IPv4/UDP/RTP packet is compressed. Assuming such a

[3]Please note that the results of the four-stages pipeline architecture is not discussed in this report

typical traffic of IPv4/UDP/RTP packets, the total number of packets that the full-HW can compress is 244000 packet/s.

The capacity is calculated in term of the number of active users that the RoHC design can serve. Assuming a voice activity of 0.5 means that a one user is sending or receiving 25 packets/s. Thus, the number of users is 9760 users.

6.5.2 Memory Bandwidth

Table 6.4 shows that the memory bandwidth needed to fetch and update the context of one user is 212.8 *kbps*. The memory bandwidth for one user is calculated as follows:

$$Memory\_bandwidth = (context\_read\_size + context\_write\_size) \times 8bit * 50packet/s \quad (6.10)$$

The total size of the context fetched from the DDR2 is 328 bytes, whereas only 204 byes are needed to be written back in a worst case when the context contains references to two IPv6 protocols, i.e, when an RTP/UDP/IPv6/IPv6 protocol stack is detected in the received packet. Figure 6.9 shows that almost 30 % of the execution time is taken by the communication overhead between the external memory and the compressor when the context is stored externally. Thus from a system level perspective, the communication with the external memory can be a limiting factor if other components share the same external memory.

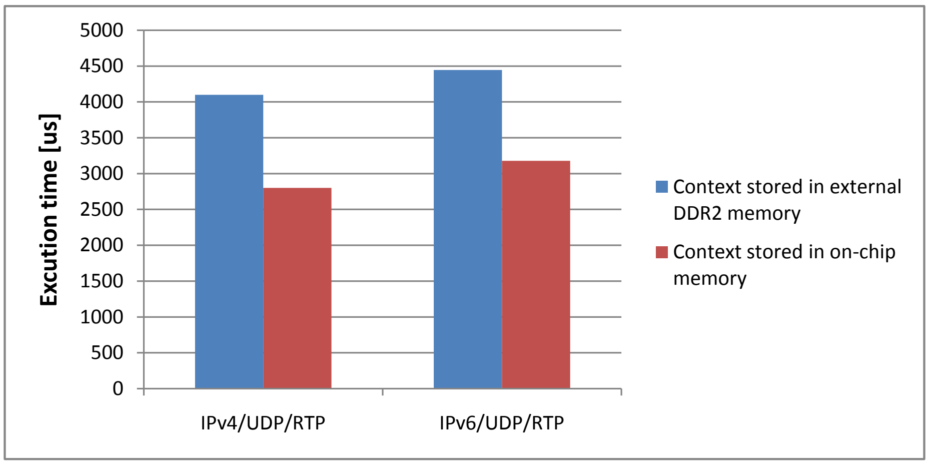


Figure 6.9: Effects of external DDR2 memory on the execution time of different protocol stacks

6.5.3 Area

The pie chart in Figure 6.10a shows that the controller stack occupies 65 % of the total design ALUTs. This makes sense since the RoHC algorithm is a control intensive algorithm, as discussed earlier. The controller state-machine consists of 211 states that control the remaining 35 % ALUTs. The controller state-machine occupies 4790 ALUTs.

The remaining 35% consists of CRC accelerators (124 ALUTs), 2 bit-packing hardware (1200 ALUTs), WLSB encoder (647 ALUTs), search LUT compnent (372 ALUTs), and arithmetic divider (233 ALUTs).

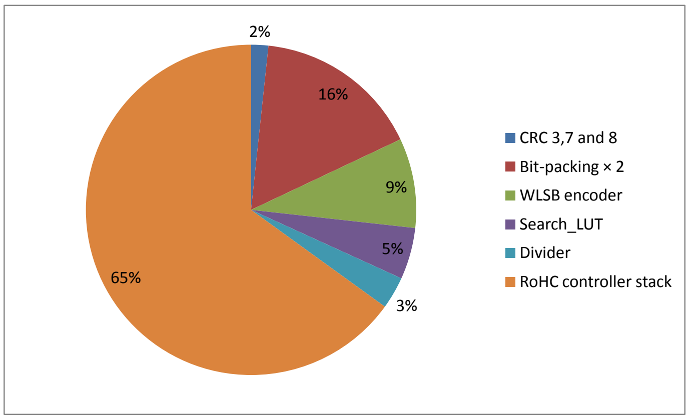
6.5.4 Power Consumption

Table 6.4 shows that the total power dissipation is 0.9 $Watt$ when clocking the design at 90 MHz. Most of the dynamic power consumption is wasted in fetching and updating the context as shown in Figure 6.10b. This is because the context is stored in the external DDR2 memory and a memory controller is needed for the communication.

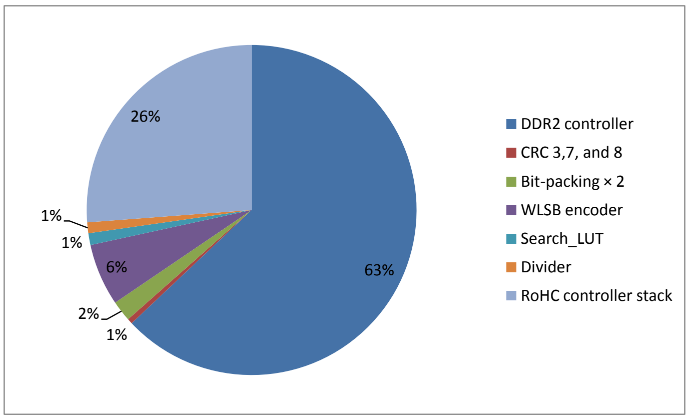
When thinking about reducing the dynamic power, a clock-gating technique can be used to switch-off the parts of the design that are in idle state. For instance, this technique can be applied to the controllers in the controller stack which are in idle state. Moreover, the dynamic power can be reduced significantly by reducing the clock frequency when less capacity is required. However, the power analysis tool in the Quartus II software shows that more than half of the total power dissipation is a leakage power.

6.5.5 Execution Time

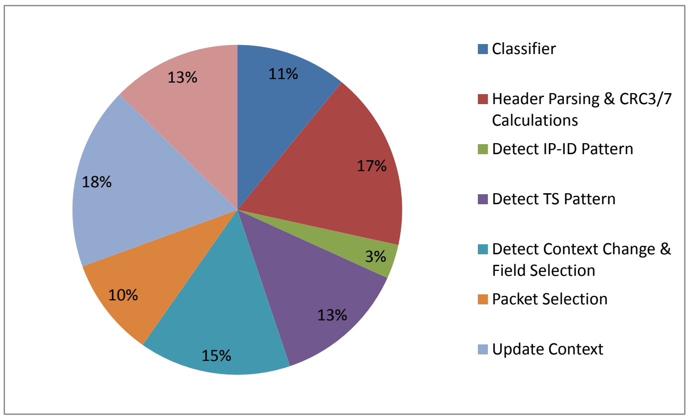
Figure 6.10c shows that the execution time almost equally distributed among the controller in the stack. Lacking a dominant component on the execution time is the reason why the increase in the performance of the HW-SW solution, presented in chapter 5, is modest. By inspecting the pie chart of the execution time, another 20-30 % increase in the HW-SW performance might be possible, if more data-path components, such as the bit-packing and the search-LUT, are implemented in HW. Since the algorithm is a control intensive algorithm and the control part checks a large amount of data, it is recommended that the communication cost is inspected when partitioning the controller into a HW-SW is desired.



(a) Area analysis



(b) Power consumption analysis



(c) Execution-time analysis

Figure 6.10: Analysis of area, power consumption, and execution-time results

Chapter 7

Comparison between Different RoHC Solutions

The proposed HW-SW and full-HW designs are implemented on the Altera Arria II GX board. By using the Quartus II software, different design metrics are extracted such as the maximum frequency with which the system can be clocked, the power dissipation, and the area in term of the number of the LUTs used. The hardware components in both solutions are described in structural VHDL and the software part in HW-SW solution is written in C language [10] and compiled on Nios II softcore micro-controller.[1]

The HW-SW solution, with a hardware partitioning degree of 10%, occupies 2578 LUTs. The full-HW solution uses three times as many LUTs as in the HW-SW design. However, the HW-SW design shows only a modest increase in the capacity metric compared to the significant improvement that the full-HW can achieve. Moreover, the power consumption in the full-HW design is around 40% less than the HW-SW design. This is because the full-HW design runs on a clock that is slower than the HW-SW design. The full-HW, HW-SW and SW-only designs are compared in Table 7.1.

In [6], a HW design is proposed targeting mobile handset application. The best processing time achieved in [6] is $40\mu s$. In [14], a hardware design is modelled in SystemC targeting wireless mesh networks. The packet processing time in [14] varies between $0.6\mu s$ to $22.6\mu s$. Therefore, the proposed Full-HW design can perform 5 to 10 times better than the designs proposed in [6] and [14] with respect to throughput. However, since the target technology, the power consumption, and the area usage of both designs in [6] and [14] are not available, it is not feasible to compare between these two designs and the proposed one with respect to any other design metric but the throughput. Figure 7.1 shows a comparison in the execution time between the proposed full-HW solution and the ones in [14] and [6].

[1]Please note that the results of the four-stages pipeline architecture is not discussed in this report

Table 7.1: Comparsion between one-stage full-HW, HW-SW, and SW-onlly implementations

Design Metrics	**Full-HW**	**HW-SW**	**SW**
Throughput[packet/second]	244k	6250	4761
Power consumption[Watt]	0.9	1.5	-
# ALUT	7477	2578	2215
# Flip-Flops	2533	1680	1680
Max frequency[MHz]	100	150	150
Memory[kB]	2	75	75
External Memory Bandwidth[kbps]	212.8	288	288

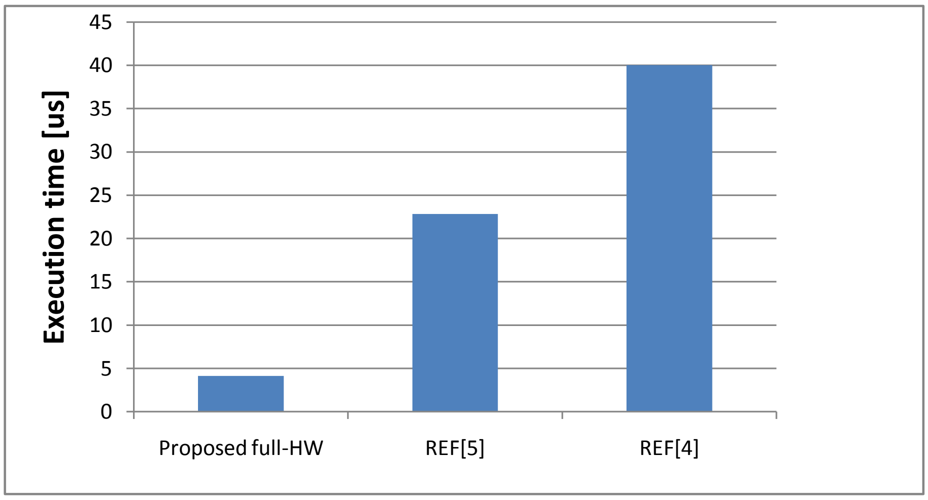


Figure 7.1: Comparison between different full-HW designs with respect to one packet execution time.

Chapter 8

Conclusions

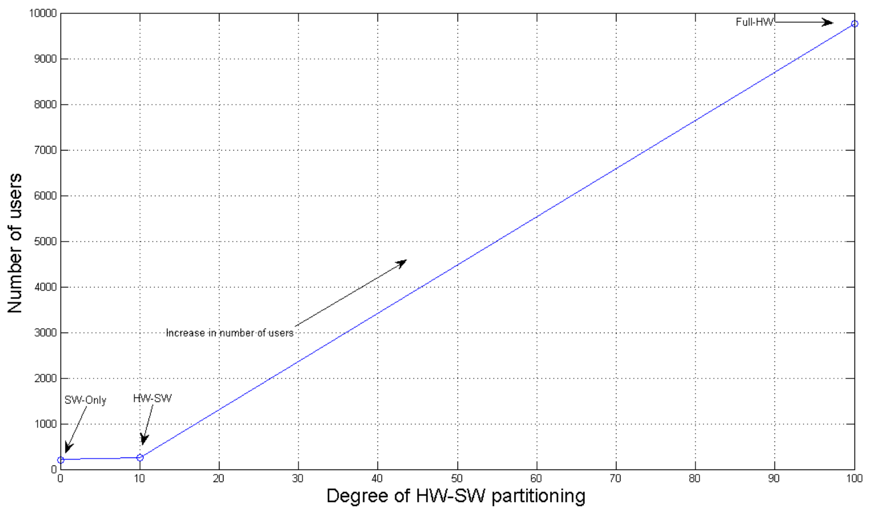


Figure 8.1: RoHC Performance Graph with respect to the HW-SW partitioning degree

This thesis work contributes with a study towards boosting the performance of the RoHC algorithm. A HW-SW and a full-HW solution are proposed and their performance metrics are compared against the SW-only implementation. These cases are important since they are the endpoints which determine the improvement in the performance with respect to the degree of the HW-SW partitioning. The number of users that the RoHC implementation can support in a base-station is plotted against the degree of partitioning as shown in Figure 8.1. An increase of 20% in the performance is obtained by a HW-SW co-design in which around 10% hardware is assisting the RoHC software. This should be compared to the 45 times increase in the performance that is achieved when a full-HW solution is used.

The plot can be used to empirically determine the percentage of the partitioning necessary to achieve a particular performance. However, when partitioning a control intensive algorithm, such as the RoHC algorithm, the communication cost might appear to be a bottleneck for achieving a particular performance. This is because partitioning a state-machine between hardware and software might add more complexity to the overall picture. Moreover, the RoHC state-machine checks a large amount of data which need to be reachable by both partitions, i.e., the hardware and the software partition.

From a system perspective, the comparison between a RoHC implementation based on a full-HW, a HW-SW design, down to a pure software-based implementation will help in deciding which methodology to choose. Based on the required capacity, the area and the power results can be used to plan the power and the area budgets for a system-level planning. When thinking about increasing the RoHC capacity, the external memory bandwidth must also be considered. The base station is required to maintain and switch between the RoHC contexts of the users. This puts hard requirements on the external memory and its bandwidth which might be shared by other components in the system.

Bibliography

[1] C. Bormann, S. BURMEI, and M. Degermark, "Rfc 3095, robust header compression: Framework and four profiles: Rtp, udp, esp and uncompressed," *IETF Standard, Internet Engineering Task Force, IETF, CH Jul*, 2001.

[2] D. Szczesny, A. Showk, S. Hessel, A. Bilgic, U. Hildebrand, and V. Frascolla, "Performance analysis of lte protocol processing on an arm based mobile platform," in *System-on-Chip, 2009. SOC 2009. International Symposium on.* IEEE, 2009, pp. 056–063.

[3] D. Taylor, A. Herkersdorf, A. Doering, and G. Dittmann, "Robust header compression (rohc) in next-generation network processors," *IEEE/ACM Transactions on Networking (TON)*, vol. 13, no. 4, pp. 755–768, 2005.

[4] "LTE Specification by 3GPP," acessed 19-August-2012. [Online]. Available: http://www.3gpp.org/LTE

[5] E. Dahlman, S. Parkvall, and J. Sköld, *4G: LTE/LTE-Advanced for Mobile Broadband.* Academic Press, 2011.

[6] D. Szczesny, S. Traboulsi, F. Bruns, S. Hessel, and A. Bilgic, "Exploration of energy efficient acceleration concepts for the rohcv2 in lte handsets," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on.* IEEE, 2011, pp. 232–237.

[7] Effnet, "Effnet rohc performance on intel core microarchitecture based processors," acessed 19-August-2012. [Online]. Available: http://www.effnet.com/19350_EFFNET_Final.pdf

[8] R. Williams, "A painless guide to crc error detection algorithms," August, 1993, acessed 19-August-2012. [Online]. Available: http://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf

[9] E. Stavinov, "A practical parallel crc generation method," *Circuit Cellar-The Magazine For Computer Applications*, vol. 31, no. 234, 2010.

[10] S. T. French space agency (CNES), Thales Alenia and V. Technologies, "Rohc-1.3.1 library," acessed 19-August-2012. [Online]. Available: http://launchpad.net/rohc/

[11] I. Quartus, “Version 11.1 handbook,” *Altera Corporation*, 2011.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “Rfc 3550, rtp: a transport protocol for real-time applications,” July, 2003, acessed 19-August-2012. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[13] J. Rey, D. Leon, A. Miyazaki, V. Varsa, and R. Hakenberg, “Rtp retransmission payload format,” *Internet Engineering Task Force, RFC*, vol. 4588, 2006.

[14] S. Jung, S. Hong, and K. Kim, “On achieving high performance wireless mesh networks with data fusion,” in *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a.* IEEE, 2007, pp. 1–8.