Low Latency Polar FEC Chain Development in Software for 5G

Final thesis presentation

Yadhunandana R. Kumaraiah (Yadhu)

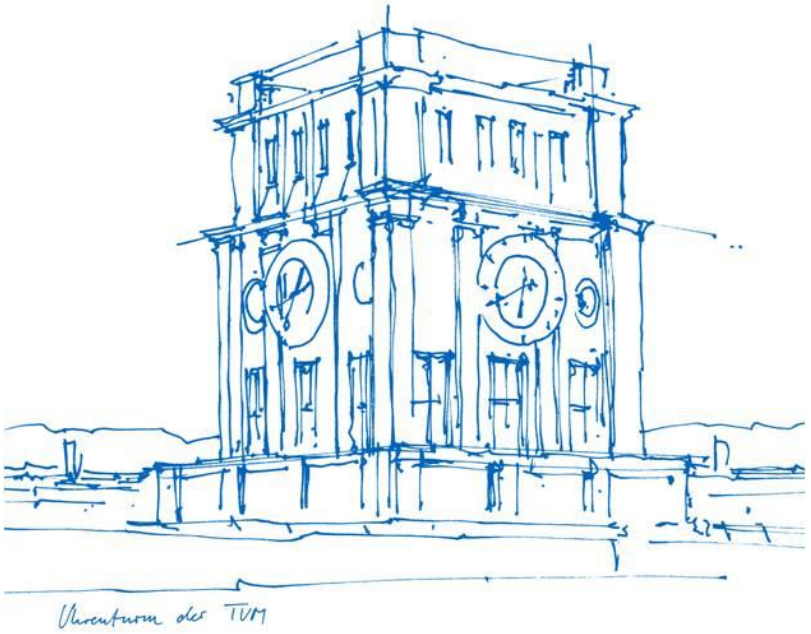
yadhu.kumaraiah@tum.de

Supervisors:

Fabian Steiner, Peihong Yuan (TUM)

Dr. Moritz Harteneck, Alexander Heinz

(Rohde & Schwarz)



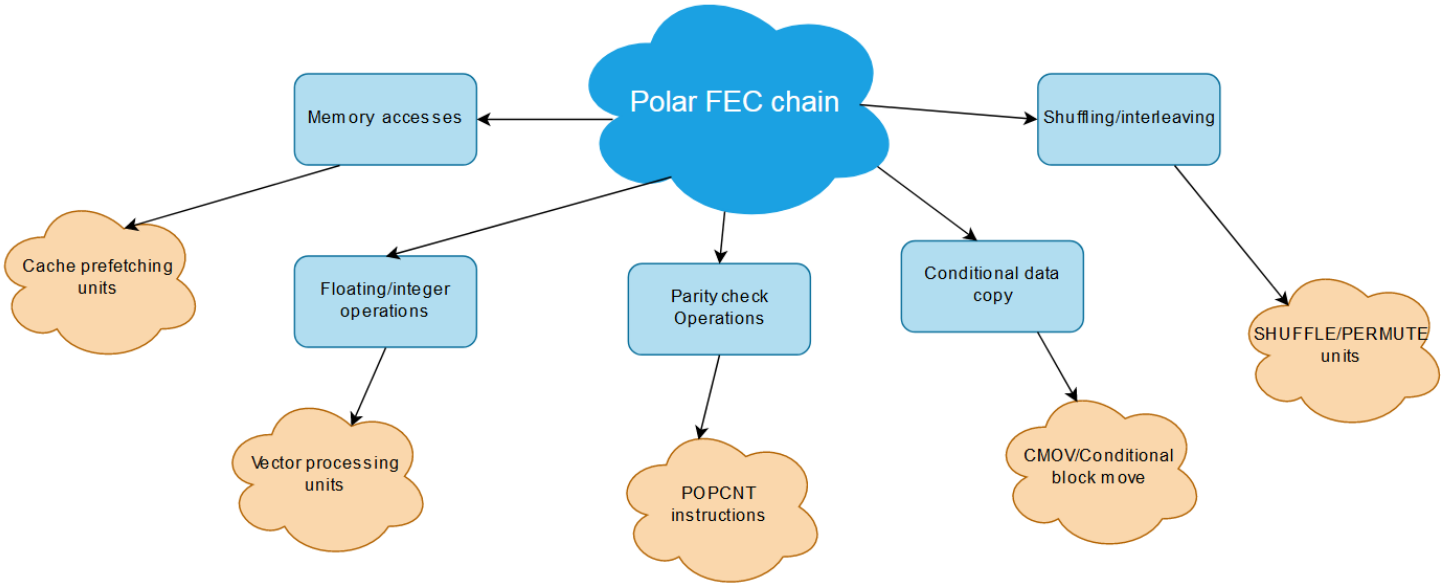
Agenda

- Motivation
- Background
- Encoding FEC chain
- Latency results : Encoding chain
- Decoding FEC chain
- Latency results : Decoding chain
- Conclusion and Outlook

Motivation

- Traditionally FEC chains are developed with FPGA's or ASIC's to achieve low latency and high throughput.
- Development/maintenance in FPGA/ASIC requires more time and expensive.
- Due to the recent advances in General Purpose Processors, it is possible to achieve required latency with software implementations without custom hardware.
- Software implementations require less development/maintenance effort and provide flexibility/ease of maintenance to device manufacturer.

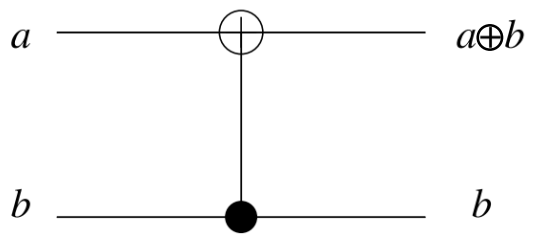
- However algorithms need to adopted/optimized to efficiently implement in software.



Background: Polar Codes

- Invented by Erdal Arıkan[1] in 2009.
- First codes to theoretically approach channel capacity for BMC channels with explicit construction.
- Basic idea is synthesizing either completely noiseless or noisy channels based on SNR.
- Information bits are transmitted in noiseless and zeros in noisy channels .
- Synthesizing channels for every SNR is complex. 5G adopted heuristic low complexity approach independent of SNR[2] .
- This construction performs sufficiently good over a large range of SNR[1]

Butterfly circuit



Encoder circuit for $N = 2$

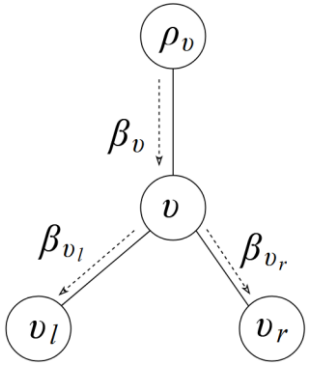
$N \to \infty$,
Fraction of good channels $\to C$

1. E. Arıkan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” IEEE Trans. Inform. Theory, vol. 55, pp. 3051–3073, July 2009.
2. β-expansion: A theoretical framework for Fast and Recursive Construction of Polar Codes. Huawei Technologies

Polar Encoding (Polar Transform)

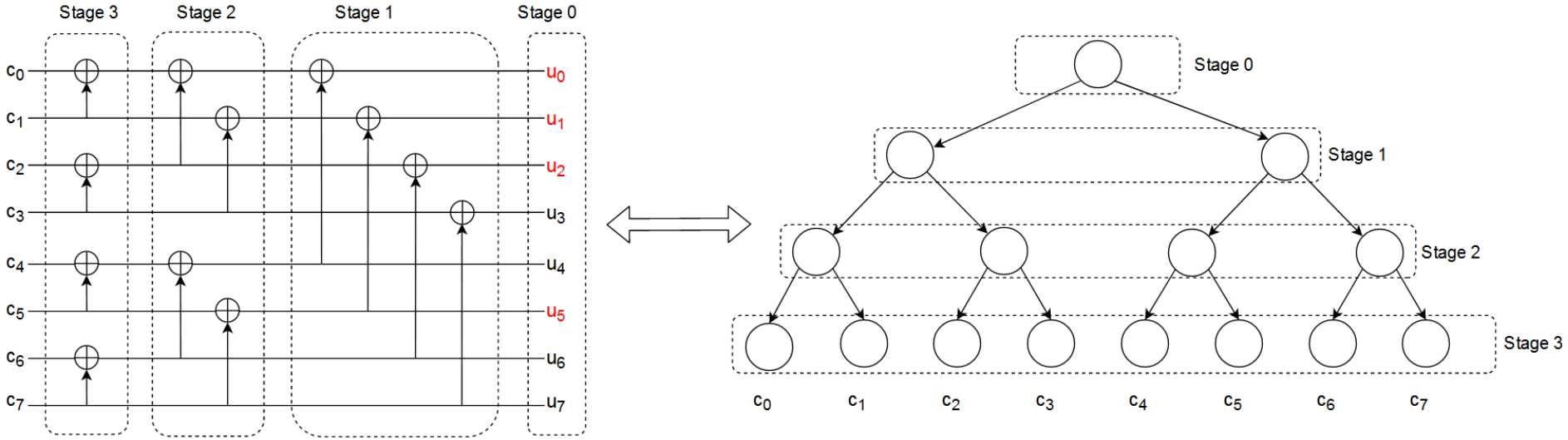
- XOR introduces correlation between code word bits.
- Bits shown in red are frozen bits (zeros).
- Node operation in encoder with vector size N_v.

$$\beta_{v_l}[i] = \beta_v[i] \oplus \beta_v[i + N_v/2] \quad 0 \le i < N_v/2$$
$$\beta_{v_r}[i] = \beta_v[i + N_v/2] \quad 0 \le i < N_v/2 \quad \text{//Superfluous copying}$$



$\oplus$ XOR operation $K = 4, N = 8$.

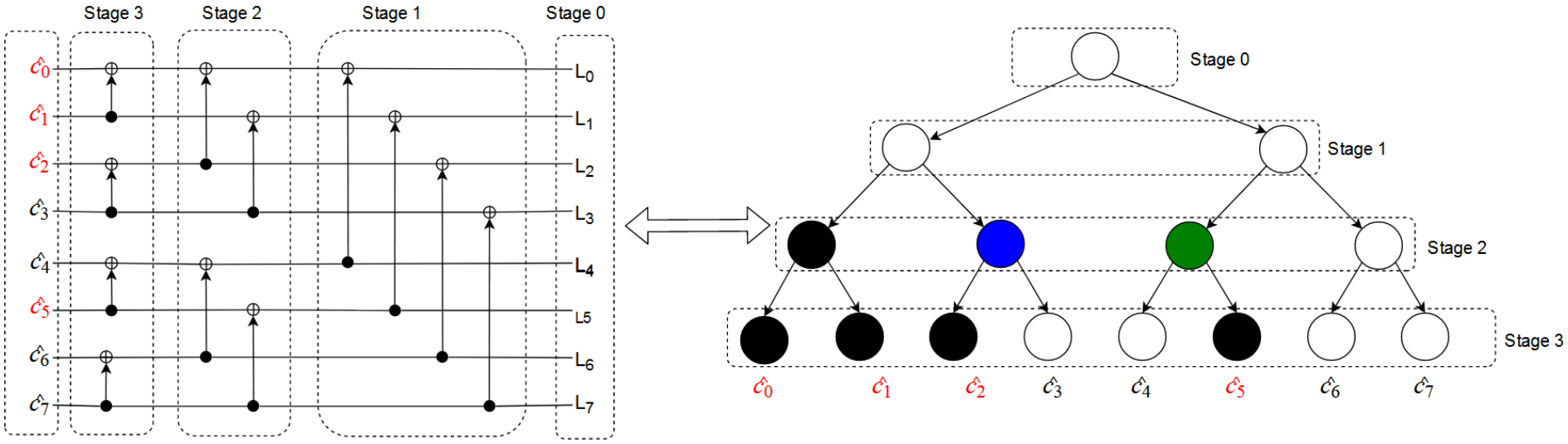
Each stage contains vector of size $Nv = 2^{N-stage}$



Decoding Polar Codes

- Decoder estimates the code word by exploiting the correlation introduced by encoder.
- Decoding involves Check Node (CN), Variable Node (VN) and bit combination operations.
- Every node in the decoder tree performs these operations.
- Decoding circuit can also be viewed as binary tree.

Each stage contains vector of size $Nv = 2^{N-stage}$



CN, VN and Bit Combination Operations

For LLR vector size N_v

- CN operation: ($\oplus$)

$$\alpha_{v_l}[i] = \text{sign}(\alpha_v[i]) * \text{sign}(\alpha_v \left[i + \frac{N_v}{2}\right]) * \min(|\alpha_v[i]|, |\alpha_v \left[i + \frac{N_v}{2}\right]|)$$

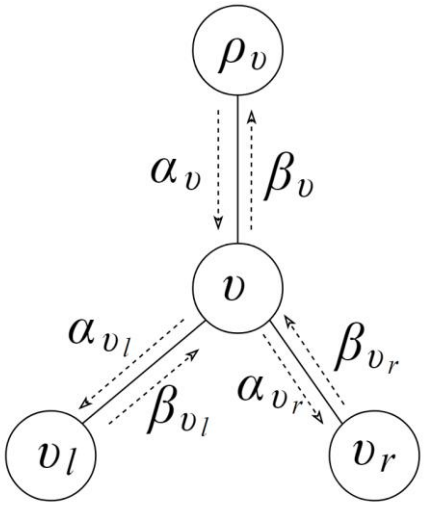
- VN operation: ($\bullet$)

$$\alpha_{v_r}[i] = \left(1 - \beta_{v_l}\right) * \alpha_v[i] + \alpha_v \left[i + \frac{N_v}{2}\right]$$

- Bit combination:

$$\beta_v[i] = \begin{cases} \beta_{v_l}[i] \oplus \beta_{v_r}[i] & 0 \le i < N_v/2 \\ \beta_{v_r}[i] & // \text{ Just copying} \end{cases}$$

Local decoder with messages



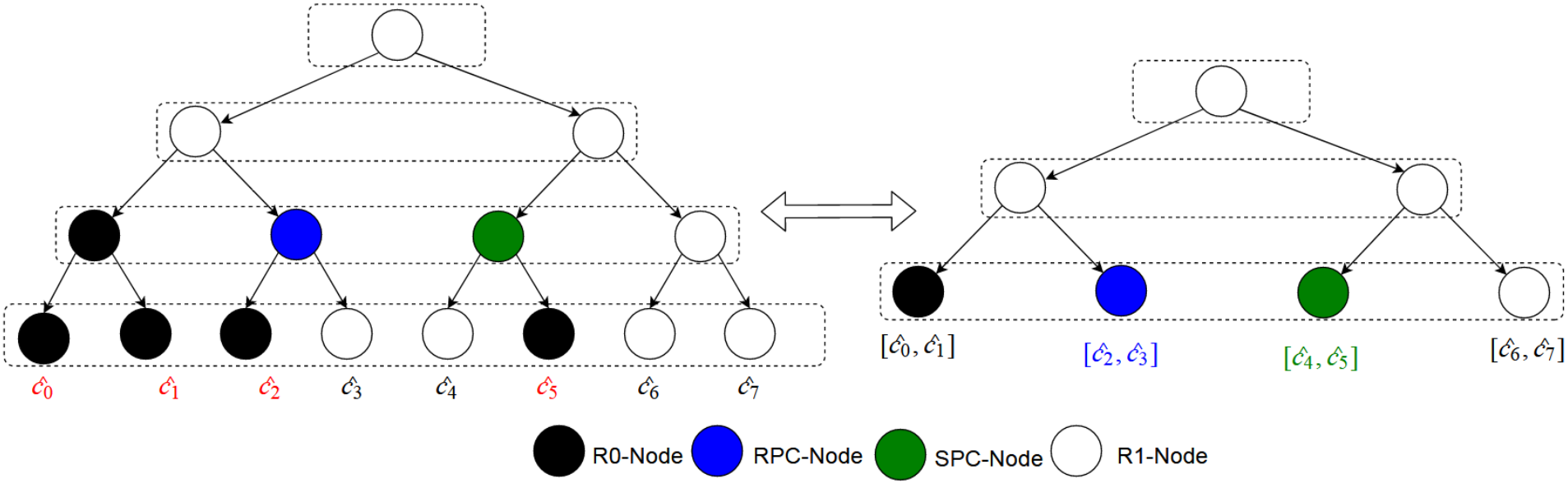
Fast-SSC Algorithm

- Fast-SSC algorithm[1] identifies special component codes from polar code which allow immediate decoding avoiding full tree traversal.
- Reduces number of CN and VN operations.
- Parallelizes the decoding operation.

- Example $frozen\_pattern = \{1,1,1,0,0,1,0,0\}$

Original decoder tree

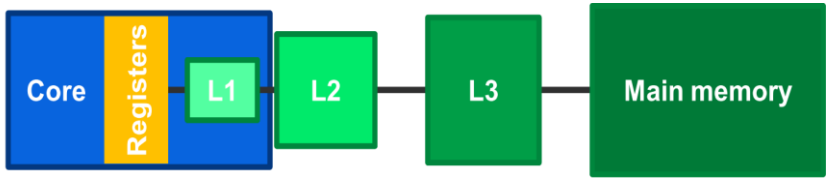
Reduced tree

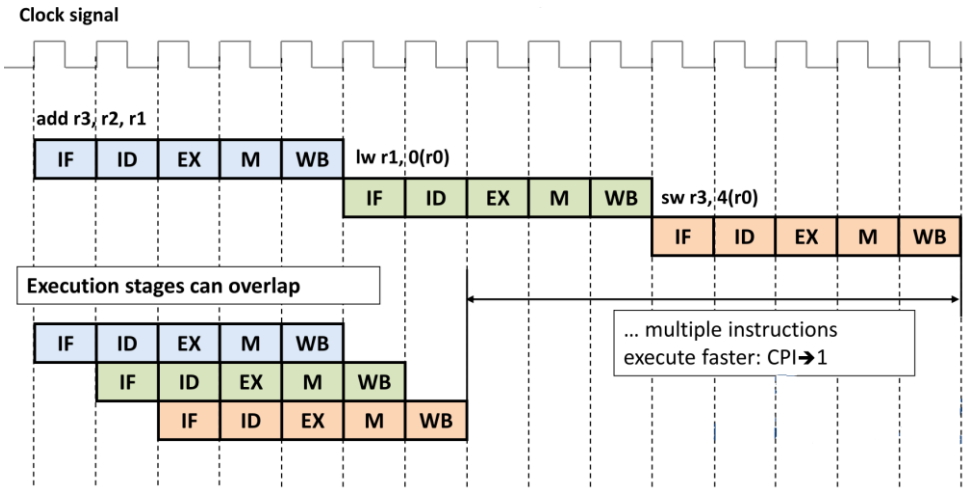


1. Fast Polar Decoders: Algorithm and Implementation. Gabi Sarkis, Pascal Giard , Alexander Vardy , Claude Thibeault, and Warren J. Gross

Processor Architecture

- Cache memory
  - Performance bottleneck in modern processors is accessing main memory.
  - Modern processors have faster memory called cache.
  - Caches reduce the average memory access latency by storing recently accessed data.
- Instruction pipelining
  - Modern processors come with advanced pipelining.
  - Pipelining increases IPC (Instructions Per Cycle).
  - Branching and cache misses create stalls in pipelining which reduce IPC. They should be reduced to achieve good performance.

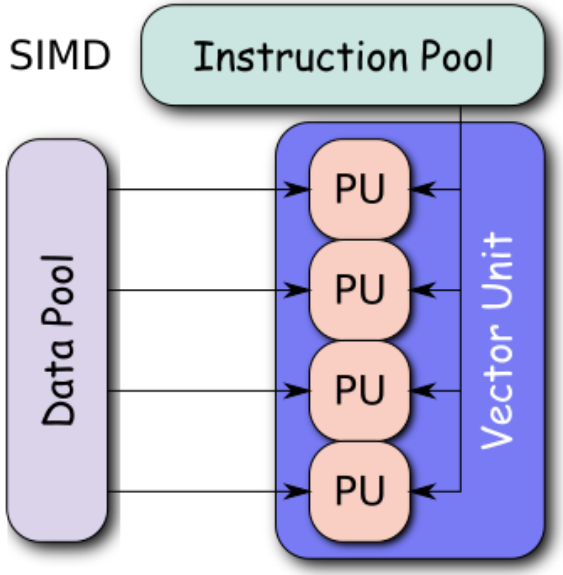




1. Reference: Ulrich Drepper, “What Every Programmer Should Know About Memory.” Red Hat, Inc.
2. A. Herkersdorf, “Chip multi core processors.” Lecture Notes, Institute for Integrated Systems, Technische Universitaet Muenchen,

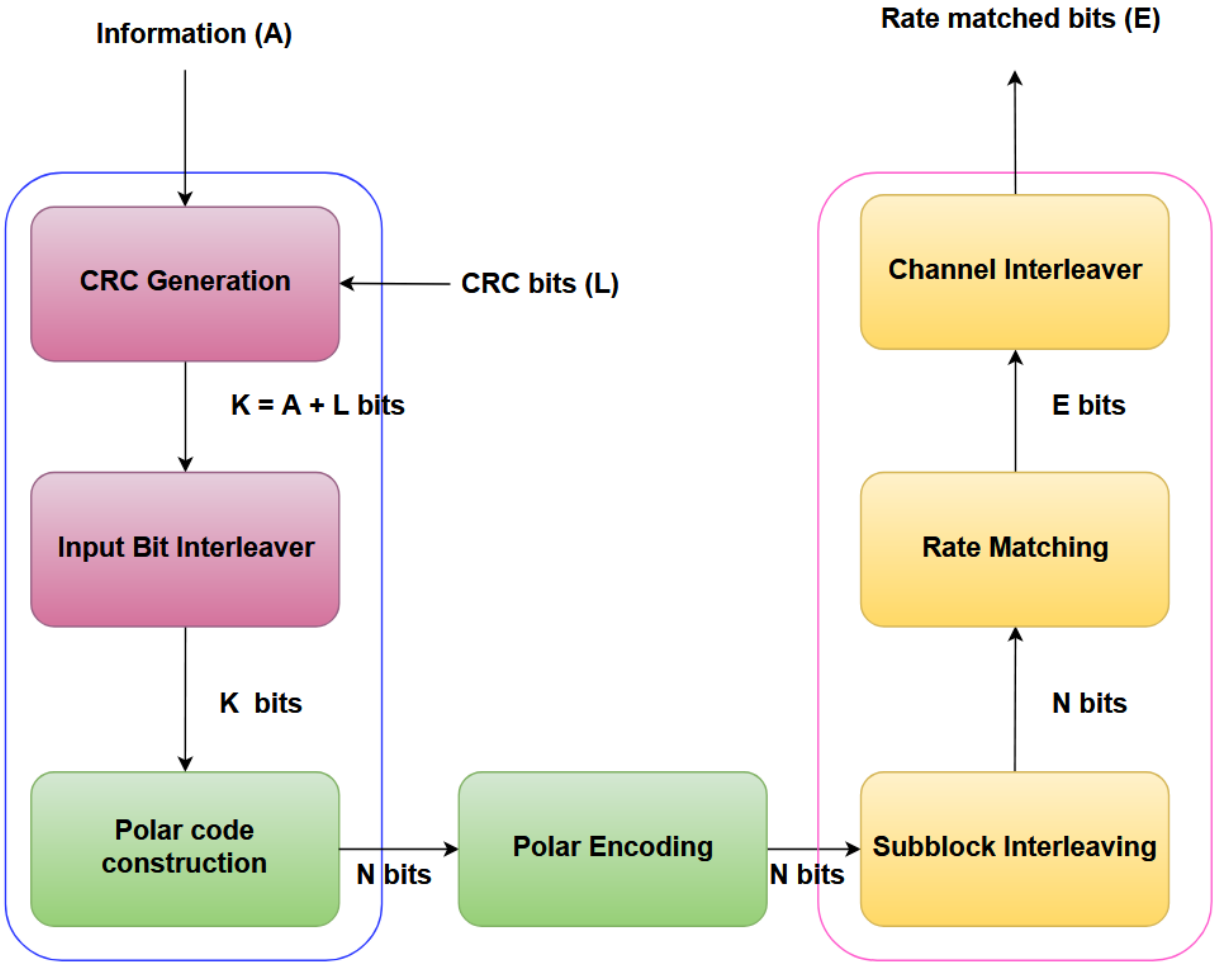
Vector Processing Units

- Process multiple data elements (vectors) in a single instruction. Incorporated in modern processors.
- Allow data parallelism, popular among video/image processing community.
- Faster encoding/decoding in communication systems can be implemented using SIMD (Single instruction multiple data) instructions.
- SIMD feature allows encoding/decoding with low latency.



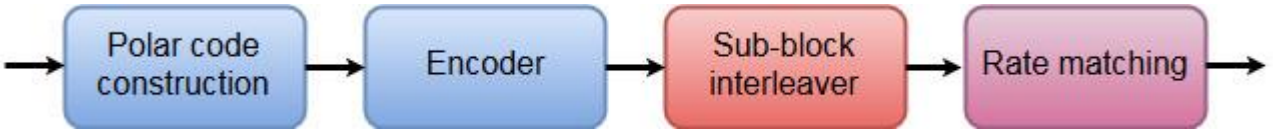
Encoding FEC Chain in 5G

- FEC chain of Physical Broadcast Channel (PBCH) and Physical Downlink Control Channel (PDCCH).

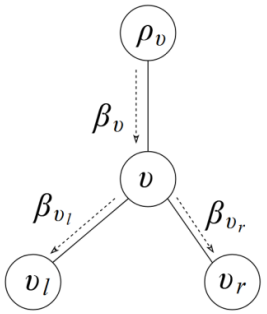


Polar Code Construction

- Reliable bit indices need to be selected by considering effect of rate matching on reliability values.
- Due to this dependency reliability indices selection process involves $(N - K)$ search and remove operations in the list, which have huge overhead. (377 µs in the functional implementation).
- Algorithm reformulated to use lookup table and mark the elements instead of removing.
- Reformulated algorithm avoids redundant copying, search and remove operations. Latency reduced to 15 µs.



Optimization of Polar Encoder

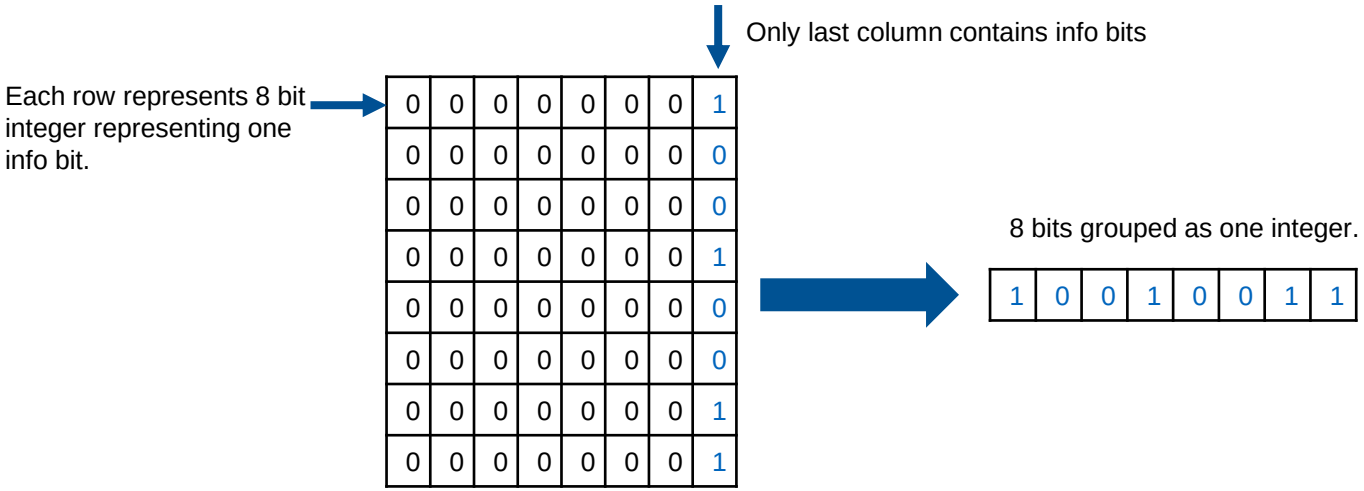


- Avoid superfluous copying:
  - Intelligent memory layout design for β_{v_r}

$$\beta_{v_l}[i] = \beta_v[i] \oplus \beta_v[i + N_v/2] \quad 0 \le i < N_v/2$$

$$\beta_{v_r}[i] = \beta_v[i + N_v/2] \quad 0 \le i < N_v/2 \;//\; \text{Superfluous copying}$$

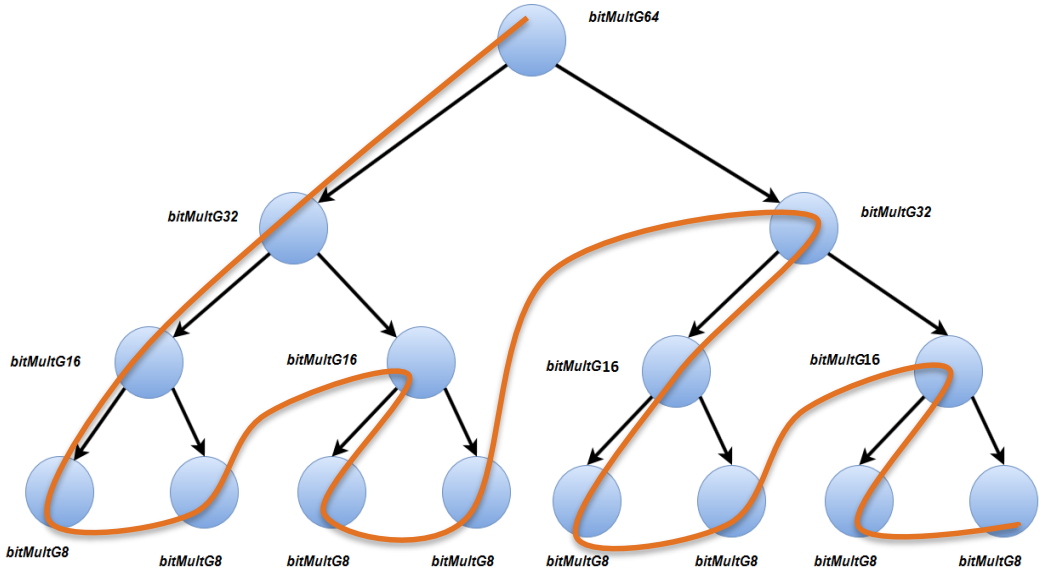
- Exploit data parallelism:
  - Naive implementation in software considers each bit as one integer. Each bit is processed sequentially.
  - Multiple bits processed in parallel by packing multiple bits to single integer. E.g. int8 = 8 info bits, int64 = 64 info bits.
  - SIMD registers are 256 bits wide, Resulting in a parallelism factor $P = 256$. SIMD processors come with fast pack/unpack instructions.



Tree Pruning and Unrolling Recursion

- Plain encoder implementation traverses till the end of tree. E.g.: $N = 1024$, encoder needs to traverse 2047 nodes.
- Pruning is done by building a lookup table and stopping encoding at $\log_2 N - 3$ level, i.e. 7 in our example.
- Number of nodes reduced to 255 from 2047 hence significantly reducing the latency.
- This optimization is applicable for hardware implementations as well.

- Recursive function has overhead since every function call requires new stack allocation and branching. Both have huge overhead.



Encoding Chain Results

Comparison with open source encoder software implementation (on AMD EPYC processor running at 1.6GHz) $N = 1024$

aff3ct[1] (No SIMD)	Current implementation
5.6 µs	0.24 µs

Encoder Latency:

Naive	Optimized
34 µs	0.24 µs

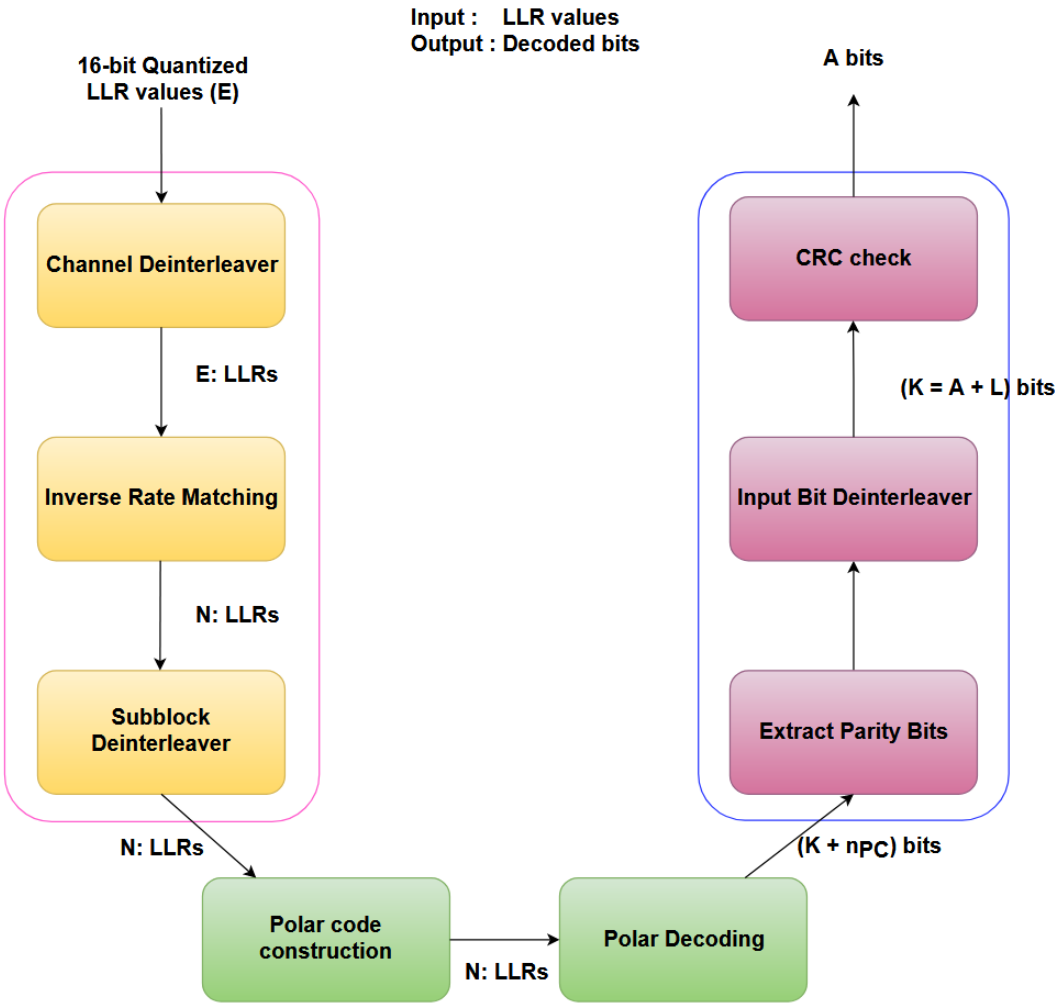
Worst case encoding FEC chain:

Naive	Optimized
451 µs	40 µs

1. A Fast Forward Error Correction Toolbox (http://aff3ct.github.io/) (No SIMD optimization)

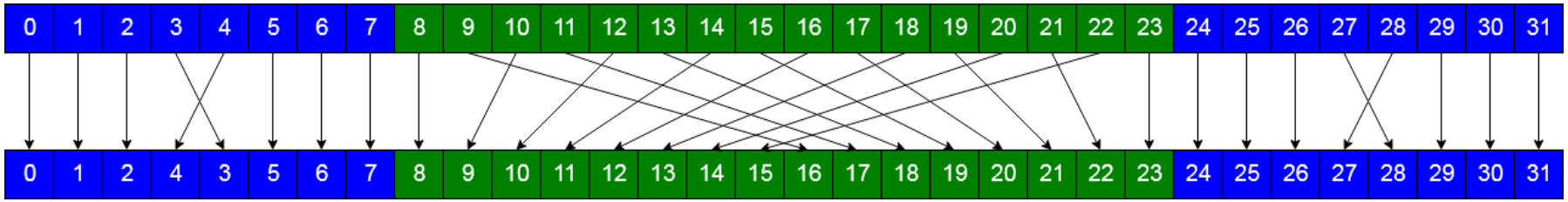
Decoding FEC Chain

- Decoding FEC chain for Uplink Control Information (UCI)



Sub-block Deinterleaver

- Inverse of subblock interleaver operation performed at the transmitter.
- Deinterleaving is performed as shown in the figure.
- Expensive operation due to huge number of division, multiplication and modulus operations and it is also sequential.
- This operation is broken down into three parts, each of them are implemented using `permute` and `blend` SIMD instructions.
- Latency reduced from 19μs to 0.47μs.



Decoding: Optimized CN, VN and Bit Combination

- Decoding operation involves CN, VN and bit combination instructions in every node of decoding tree.
- Efficient implementation of these operations is critical for achieving low latency.
- CN, VN and bit combination operations are vectorized.
- Sign multiplication in CN operation is reduced to bitwise negation and implemented using SIMD instructions.
- In VN operation multiplication based on β_{v_l} is reformulated using bit wise negation.
- Bit combination is performed with `SIMD XOR` instructions.

Avoiding superfluous copying:

- Intelligent memory layout design for β_{v_r} so that it is part of β_v without explicit copying.

- CN operation: ($\oplus$)

$$\alpha_{v_l}[i] = sign(\alpha_v[i]) * sign(\alpha_v \left[i + \frac{N_v}{2}\right]) * \min(|\alpha_v[i]|, \alpha_v \left[i + \frac{N_v}{2}\right])$$

- VN operation:

$$\alpha_{v_r}[i] = \left(1 - \beta_{v_l}\right) * \alpha_v[i] + \alpha_v \left[i + \frac{N_v}{2}\right]$$

- Bit combination:

$$\beta_v[i] = \begin{cases} \beta_{v_l}[i] \oplus \beta_{v_r}[i] & 0 \le i < N_v/2 \\ \beta_{v_r}[i] & \text{//superfluous copy} \end{cases}$$

Packing Frozen Pattern

- Packing multiple frozen bits allows efficient identification of component codes.
- Frozen pattern is passed in packed bit format to decoder.
- Identification component codes is performed at run time.
- Packing of multiple bits together allows identification of component codes in a single comparison instruction.
- Reduces the expensive branch instructions and exploits data parallelism.

- For frozen pattern at child node = {1,1……..1,1} //256 values.

Naive way

```
bool isRateZeroNode(int8_t frozenPattern[])
{
    bool rateZeroNode = true;
    for(auto i = 0; i < 256;i++) {
        if(frozenPattern[i] != 1) {
            rateZeroNode = false;
            break;
        }
    }
    return rateZeroNode;
}
```

Efficient SIMD way

```
template<>
inline bool isRateZeroNode(uint64_t s[]) {

    __m256i temp1 = _mm256_loadu_si256 ((__m256i*)s);
    __m256i temp2;
    temp2 = _mm256_set1_epi8 ((char)0xFF);

    __m256i pcmp = _mm256_cmpeq_epi64 (temp1, temp2);

    unsigned bitmask = _mm256_movemask_epi8(pcmp);

    return (bitmask == 0xffffffffU);
}
```

Decoding R0 and R1 Nodes

- Decoding R0 code
  - R0 node contains all frozen bits.
  - Decoded bits are set to zero. Further tree traversal isn't necessary.

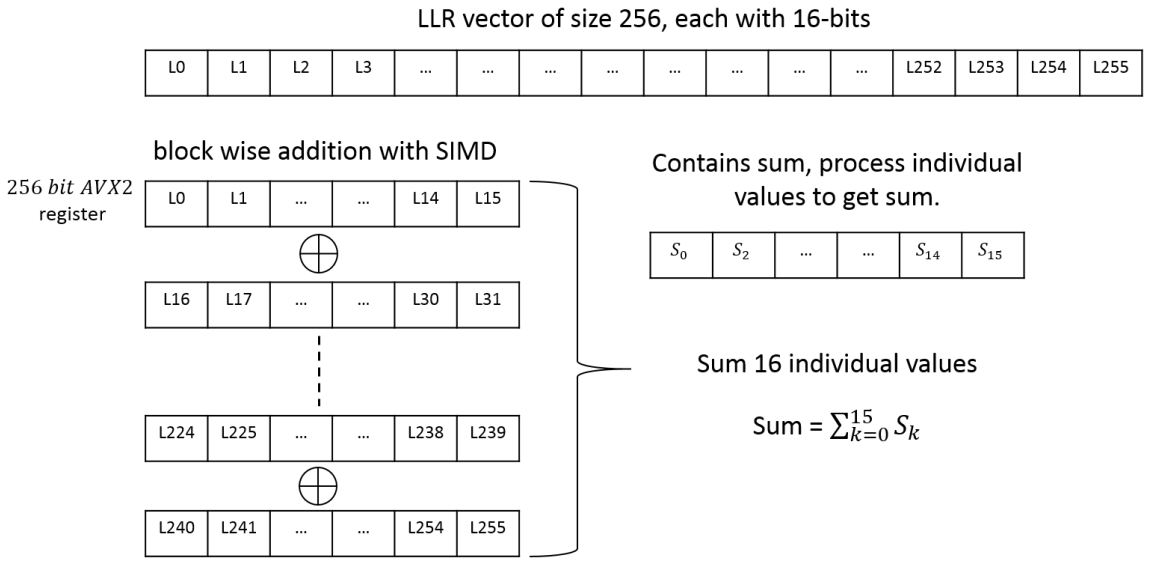
- Decoding R1 code
  - R1 node contains all information bits.
  - No extra information is gained by traversing full tree.
  - Simple hard decision decoding is performed at this node.
  - Decoding performed by applying threshold detection and polar transform.

- To speedup R1 node decoding, threshold detection and polar transform is vectorized with a parallelism factor of 16 ($P = 16$)
- Processes vectors of size 16 using SIMD instructions. Namely with `vpcmpeq` and `vpxor`.

Decoding RPC Node

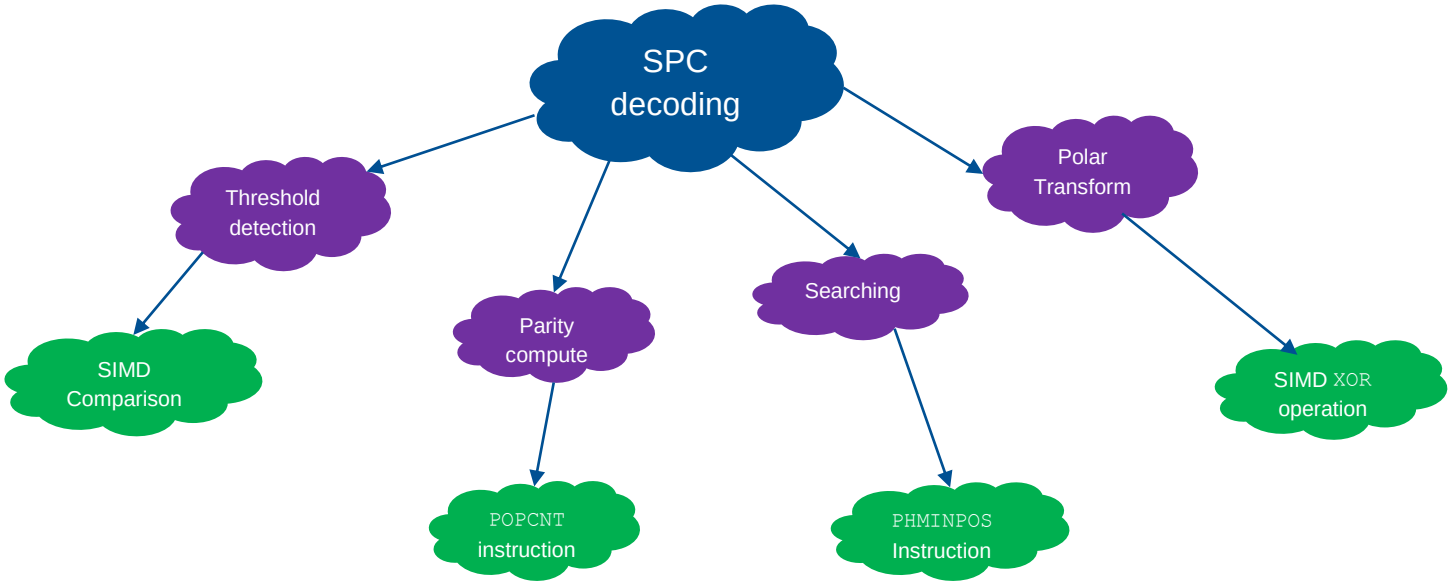
- RPC node's right most descendent contains information and all remaining bits are frozen.
- Decoding operation involves summing of all the elements of LLR vector and deciding the bit based on the sum.
- Result is stored at information bit position and zeros are stored in frozen bit positions.
- Summation is calculated block wise with AVX2 vector instructions.

$$\beta_v[i] = \begin{cases} 0, & \sum_j \alpha_v[j] \geq 0 \\ 1, & otherwise \end{cases}$$



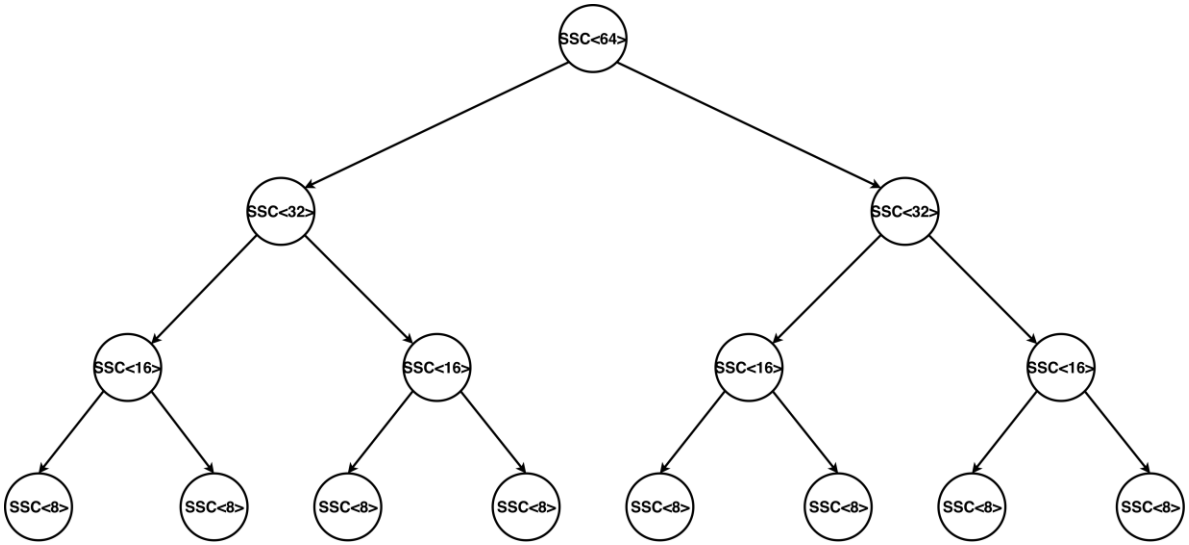
Decoding SPC Node

- SPC node's left most descendent is a frozen bit and remaining positions contain information bits.
- Decoding involves threshold detection, parity calculation, searching minimum magnitude LLR and polar transform.
- Block wise threshold detection with SIMD comparison instruction.
- Parity calculation with bit packing and `POPCNT` instruction.
- Searching is performed with another SIMD instruction `phminpos`.
- Finally optimized polar transform function is called.



Decoder Tree Pruning and Unrolling Recursion

- Tree pruning
  - Decoding latency can be further reduced by intelligently pruning the decoder tree. Pruning irrespective of frozen pattern.
  - Latency reduction comes at the cost of increased BLER.
  - High SNR and low code rate scenarios this method can be used. Level of pruning and BLER can be dealt as trade-off.
- Unrolling recursion
  - Recursion suits hardware implementation. However in software, it has a huge overhead.
  - Decoder implementation is unrolled using templates concept of C++.
  - With templates, compiler automatically generates the code for different vector sizes.



CRC Calculation

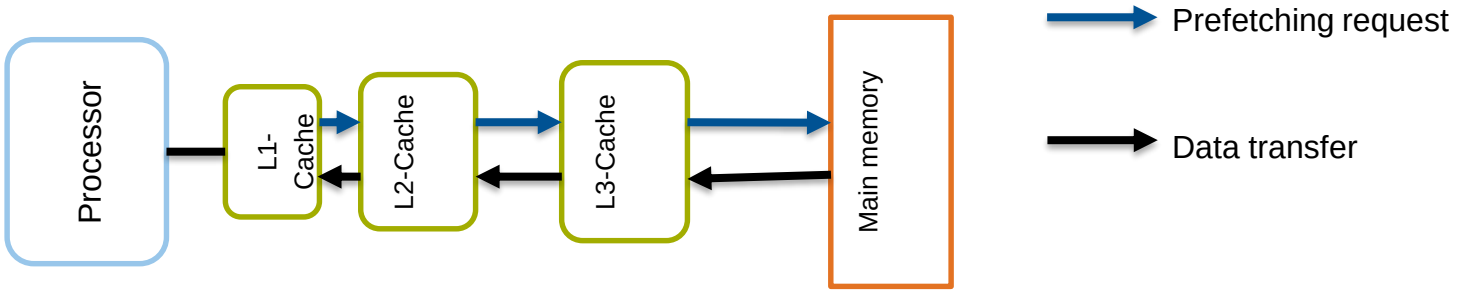
- Encoding FEC chain need to calculate *CRC24* for downlink control information transmission.
- Decoding FEC chain calculates *CRC6* or *CRC11* for CRC check.
- CRC calculation is one of the significant latency contributor to both encoding and decoding FEC chains.
- Processing individual bits for CRC calculation is very inefficient.
- Extended the popular CRC algorithm[1] to calculate CRC24, CRC6 and CRC11 .
- In this algorithm, 8 bits are processed in parallel rather than bit by bit,
- Reduced CRC calculation latency from 8 µs to 0.8 µs in FEC chain.

1. D.V. Sarwate, “Computation of Cyclic Redundancy Checks via Table Lookup,”Comm. ACM, vol.31, no. 8, pp. 1008-1013,Aug. 1988.

Miscellaneous Optimizations

- Replaced multiplication/division and modulus operations with bitwise operations which achieve the same result
- Implemented approximate versions logarithm and exponential functions to reduce the number of floating point multiplications.
- Reduced the usage of jump functions to avoid flushing of the instruction pipeline, instead latest instruction extension `CMOV` is used.
- Used the compiler optimization primitives for better instruction scheduling.

- Cache Prefetching
  - VN, CN and bit combination operations fetch a block of memory and access pattern is predictable.
  - Memory access latencies reduced by fetching cache well in advance.
  - Cache line fetched with `PREFETCH` instruction provided by `3dnow` extension of EPYC processor.



Decoding Chain Results

Comparison with state of the art polar decoder software implementation (in AMD EPYC processor at 1.6 GHz) $N = 1024$

Decoder latency compared to state of the art

[1]*	This work
8 µs	5 µs

*Scaled according to frequency

State of the art : 8-bit LLRs
This work: 16-bit LLRs

Decoder latency:

Naive	Optimized
283.4 µs	5 µs

Worst case latency decoding FEC chain:

Naive	Optimized
391 µs	40 µs

1. P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, “Low-latency software polar decoders,” J. Signal Process. Syst., vol. 90, pp. 761–775, May 2018.

Conclusion

- Both encoding and decoding FEC chains are efficiently implemented, achieved latency requirements through both algorithmic and software optimizations
- Optimized implementation exploits modern features. Namely SIMD, Cache prefetching etc.
- Achieved latency reduction of 10x.

Outlook

- CRC aided Successive Cancellation List (*CA-SCL*) algorithm is also implemented. However it is not optimized.
- *CA-SCL* algorithm has approximately 1.5dB gain over *fast-SSC* algorithm for N = 2048 and list size of 8[1].
- Future work: Extend decoding FEC chain with optimized *CA-SCL*.

1. I. Tal and A. Vardy, “List decoding of polar codes,” IEEE Transactions on Information Theory, vol. 61, pp. 2213–2226, May 2015

Encoding Chain results:

Encoder Latency:

Naive	Optimized
34 µs	0.24 µs

Worst case encoding FEC chain:

Naive	Optimized
451 µs	40 µs

Decoding Chain results:

Decoder latency compared to state of the art

[1]* (8-bit LLR)	This work (16-bit LLR)
8 µs	5 µs

Improvement in decoder latency:

Naive	Optimized
283.4 µs	5 µs

Worst case latency decoding FEC chain:

Naive	Optimized
391 µs	40 µs

Thank you

Questions

1. P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, “Low-latency software polar decoders,” J. Signal Process. Syst., vol. 90, pp. 761–775, May 2018.