SOFTWARE IMPLEMENTATION OF NEAR-ML SOFT-OUTPUT MIMO DETECTION

Teo Cupaiuolo, Daniele Lo Iacono
Advanced System Technology
STMicroelecronics Italy
teo.cupaiuolo@st.com, daniele.loiacono@st.com

ABSTRACT

The continuous emerging of new communication standards is pushing towards the introduction of the Software Defined Radio (SDR) concept. SDR is enabled by performing computational intensive task in software rather than using dedicated hardware. Within the SDR framework, Soft-Output (SO) Multiple-Input Multiple-Output (MIMO) detection is still a major challenge, which only few papers have dealt with so far. In this paper we describe the implementation of the Layered ORthogonal Lattice Detector (LORD) SO MIMO detector on the programmable Block Processing Engine (BPE). Results show that real-time MIMO detection can be achieved using a cluster of four BPEs running at 350 MHz (65 nm STMicroelectronics CMOS technology) and delivering up to 150 Mbit/s for the 64-QAM modulation, 2×2 antennas configuration.

1. INTRODUCTION

Multiple antennas wireless communications currently enjoy great popularity because of the demand of high data rate such as multimedia services. MIMO transmission consists of the simultaneous transmission of T complex symbols using T transmit antennas. MIMO systems take advantage of multi-path propagation to increase the diversity gain and enhance channel capacity on frequency-selective fading channels when operating in a rich scattering environment.

Among the others, a significant example of a system endorsing MIMO combined with OFDM is provided by the next generation Wireless Local Area Networks (WLANs), see e.g., the IEEE 802.11n standard [1].

Despite the increasing interest in commercial baseband software implementation, available literature on Soft-Output (SO) MIMO detection mainly targets Application Specific Integrated Circuit (ASIC) design [2][3][4]. Although it remains a major challenge due to high computational complexity, a few examples already exist implementing MIMO over programmable architectures.

IMEC has been the first, and indeed one of the few disclosing a software implementation of a full 2×2 MIMO OFDM transceiver. IMEC proposes a platform embedding the ADRES processor, a coarse-grain Application Specific Instruction Set Processor (ASIP) specifically designed for communications [5]. Similar work has been disclosed by ETH using a dual-core platform based on the fine-grain ASPE processor [6]. Both solutions target the Minimum Mean Square Error (MMSE) algorithm, which is highly sub-optimal when compared to near-ML detection [7].

Recently IMEC enhanced the ADRES platform so as to support near-ML detection for a 2×2 IEEE 802.11n [8]. The platform evolved from a 4-way SIMD to a 16-way SIMD to deliver a maximum throughput enabling real-time 64-QAM detection.

This paper presents a software implementation of the Layered Orthogonal Lattice Detector (LORD), a SO near-ML MIMO detection algorithm [7]. The algorithm has been mapped on the Block Processing Engine (BPE), a fine-grain vector processor specifically optimized for intensive wireless communications [9].

The paper is organized as follows: Section 2 details the target system model and recalls the LORD algorithm; Section 3 presents the BPE core; Section 4 describes the mapping of LORD on the BPE; Section 5 summarizes the results.

2. MIMO DETECTION

2.1. SYSTEM MODEL

In order to simplify the notation we consider a frequency non-selective MIMO communication system with T transmits and R receives antennas. For OFDM systems, like those of interest for 802.11n WLANs, the following equations are to be intended valid per sub-carrier in frequency domain.

The signal received at each antenna is therefore a superposition of the T transmitted signals corrupted by multiplicative fading and additive white Gaussian noise. The complex path gains are samples of zero mean Gaussian

Random Variables (RV) with variance $\sigma^2 = 0.5$ per dimension. Fading processes for different transmit and receive antenna pairs are assumed to be independent. Complex gains are assumed constant over the duration of a codeword and vary independently from one codeword to another (i.e. quasi-static block fading). Ideal Channel State Information (CSI) at the receiver is assumed (i.e. the R×T channel matrix **H** is perfectly known). The transmitted signal can be represented as a vector **X** of size T×1, where the t-th symbol s_t taken from a generic M^2-QAM constellation is transmitted by the t-th antenna. Under these assumptions, the received R×1 vector **Y** is given by:

$$\mathbf{Y} = \sqrt{\frac{E_s}{T}}\mathbf{H}\mathbf{X} + \mathbf{N}, \quad (1)$$

where E_S is the total per symbol transmitted energy (under the hypothesis that the average constellation energy is $|s_k^2| = 1$) and **N** is the noise vector of size R×1, whose elements are samples of independent circularly symmetric zero-mean complex Gaussian RVs with variance $N_0/2$ per dimension. The signal-to-noise-ratio (SNR) per receive antenna is E_S/N_0.

ML detection over a MIMO channel corresponds to finding the transmitted sequence $\overline{\mathbf{X}}$ which minimizes the square norm of the error matrix:

$$\overline{\mathbf{X}} = \arg\min_{\mathbf{x}} \left\| \mathbf{Y} - \sqrt{\frac{E_s}{T}}\mathbf{H}\mathbf{X} \right\|^2. \quad (2)$$

Equation (2) can be solved by performing the exhaustive search of M^{2T} sequences, where M is the modulation order of a generic M^2-QAM constellation. This results into a prohibitive complexity for growing T.

2.2. THE LORD ALGORITHM

Prior to the detection stage, QR decomposition is applied to the channel matrix **H**, generating an orthonormal matrix $\mathbf{Q}^t$ and an upper triangular matrix $\mathbf{R}^t$ as $\mathbf{H}^t = \mathbf{Q}^t\mathbf{R}^t$.

The index t refers to symbol sequence permutations where the t-th layer is taken as reference layer (the terms layer and transmit antenna will be used interchangeably throughout this paper); more specifically, each permutation has to differ from the others by the complex symbol placed in the t-th position in the complex sequence **X**, corresponding to the t-th I and Q couple in the real sequence $\mathbf{x}^t$. Multiplying (1) by $(\mathbf{Q}^t)^T$:

$$\tilde{\mathbf{Y}}^t = (\mathbf{Q}^t)^T\mathbf{Y}^t = \mathbf{R}^t\mathbf{X}^t + \tilde{\mathbf{N}}^t. \quad (3)$$

The noise vector $\tilde{\mathbf{N}}^t$ has still independent components and equal variances.

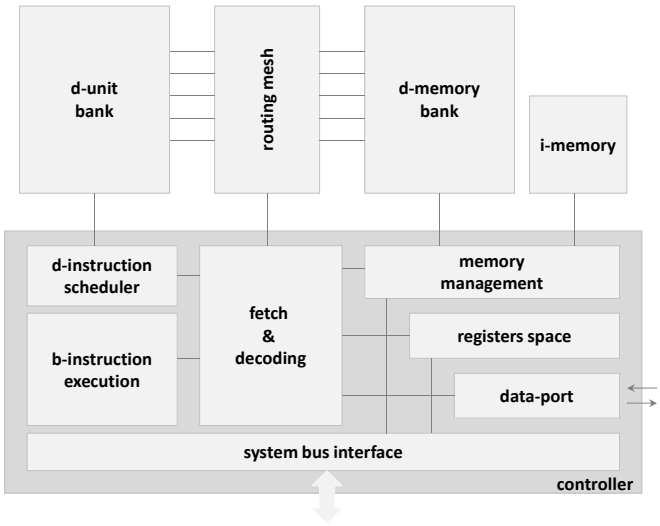


Fig. 1. The BPE template architecture.

From the above expression, the minimization problem (2) translates to:

$$\hat{\mathbf{X}}^t = \arg\min_{\mathbf{x}} \left\| \tilde{\mathbf{Y}}^t - \mathbf{R}^t\mathbf{X}^t \right\|^2. \quad (4)$$

After the QR decomposition, the ML demodulation (4) can be evaluated according to the so-called max-log approximation. The Log-Likelihood Ratio (LLR) of the bit $b_{T,k}$ can be expressed as [7]:

$$L(b_{T,k}|\tilde{\mathbf{y}}^t) = \min_{\{\tilde{x}_{2T-1},\tilde{x}_{2T-1}\}\in S(k)_T^-} D_{ED}^t[\hat{\mathbf{x}}(\tilde{x}_{2T-1},\tilde{x}_{2T-1})] - \min_{\{\tilde{x}_{2T-1},\tilde{x}_{2T-1}\}\in S(k)_T^-} D_{ED}^t[\hat{\mathbf{x}}(\tilde{x}_{2T-1},\tilde{x}_{2T-1})], \quad (5)$$

where D_{ED}^t is the Euclidean Distance (ED) metric:

$$D_{ED}^t(\mathbf{x}) = \left\| \tilde{\mathbf{y}}^t - \mathbf{R}^t\mathbf{x} \right\|^2. \quad (6)$$

In (5), the following notation is used: M_c-bit transmitted symbols belong to a M^2-QAM complex constellation; $\hat{\mathbf{x}}(\tilde{x}_{2T-1},\tilde{x}_{2T})$ denotes the sequence obtained by grouping a candidate value $(\tilde{x}_{2T-1},\tilde{x}_{2T})$ of the I and Q couple of the reference layer complex symbol X_T and the (2T−2) I and Q estimates of the T−1 non-reference layer symbols determined through spatial Decision Feedback Equalization (DFE) starting from such candidate value; $b_{T,k}$ are the bits mapped onto X_T having bit index k = 1,…,M_c; $S(k)_T^+$ and $S(k)_T^-$ represent the sets of symbols of the reference layer having $b_{T,k} = 1$ and $b_{T,k} = 0$, respectively [7].

It should be recalled that the LORD demodulation method requires to consider all the constellation symbols as candidate symbols for each reference layer and then minimizes the ED metrics over the sequences **X** wherein a given bit value is 1 or 0.

3. THE BLOCK PROCESSING ENGINE

The template architecture of the BPE is shown in Fig. 1

The controller performs fetch, decoding and scheduling of the instructions. Two types of instruction exist: basic scalar instructions (hereafter called b-instruction) mainly devoted to flow control and data access configuration and dedicated vector instructions (d-instruction) performing intensive data processing. While b-instructions are locally executed, d-instructions are executed on the customizable d-unit bank. Depending on data dependencies and resources availability, units can be scheduled to run in parallel. Vectors are allocated on the d-memory bank, a set of static memories allowing fast and parallel access to data.

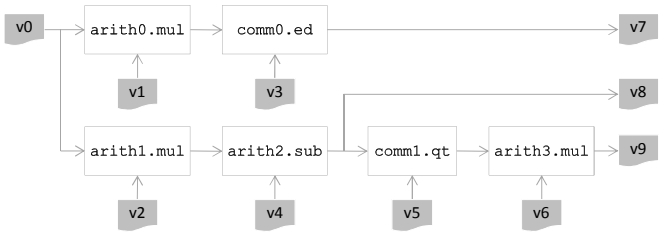
Interconnection between d-memory bank and d-unit bank is guaranteed by the routing mesh, which is run-time configured by the controller on an instruction-by-instruction basis. To further optimize the data exchange between units, the routing mesh supports instruction pipelining through direct connection between units.

3.1. VECTORS MANAGEMENT

D-instructions act on vectors allocated on the d-memory bank. Each memory can hold several vectors, with the obvious limitation that vectors allocated on the same memory cannot be accessed concurrently. Vectors size can be as large as the size of available memories.

Once the size and the position of a vector within the memory have been defined, the addressing scheme can be specified by the programmer on an instruction basis.

Vector elements can be accessed performing intra-vector permutation according to pre-defined or user-defined schemes, such as decimating or interpolating by an arbitrary factor, reversing the order of the elements, applying well known shuffling patterns like matrix transpose, FFT re-ordering, Gray mapping.

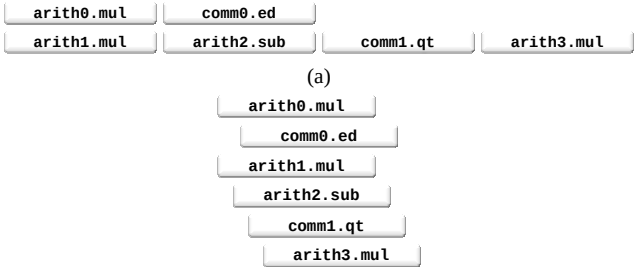


(a)

```
arith0.mul(v0,v1);
arith1.mul(v0,v2)
v7 = comm0.ed(arith0,v3)
v8 = arith2.sub(v4,arith1)
comm1.qt(arith2,v5)
v9 = arith3.mul(comm1,v6)
```

(b)

Fig. 2. Macros implementation: block diagram (a) and assembly code (b).



(a)

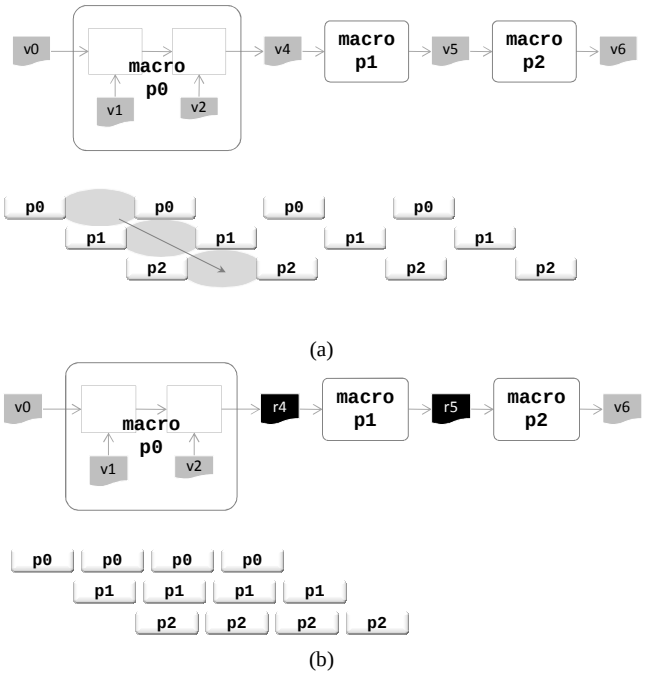
(b)

Fig. 3. Macro timing diagram when using d-instruction parallelism only (a) and both parallelism and pipelining (b).

3.2. RESOURCES ALLOCATION

Dealing with very long vectors, d-instructions are typically asked to process large amount of data, thus consuming several clock cycles to complete. While this has the benefit of considerably shortening the program size, it requires a complex semaphore mechanism between the controller and the d-unit bank to manage resources allocation and to prevent stalling.

The controller fetches and schedules instructions one after another until one of them requires resources that have been already allocated, as can be the case for a d-memory or



(a)

(b)

Fig. 4. Pipeline builder (a) and pipeline optimization through the use of memory aliasing (b).

another d-unit. It then waits until the execution of the instructions using those resources has been completed. This mechanism has the major benefit of being agnostic with respect to the latency of each d-unit, requiring the controller to be notified only when a resource has been released. A side benefit of such policy is that b-instructions executed right after the scheduling of d-instructions do not cause additional delay. The latter consideration inherently suggests that maximum efficiency can be reached only using vectors large enough to absorb b-instruction execution.

3.3. UNITS PIPELINING

As previously stated, routing mesh can be instructed to directly connect units. Pipelined processing is the key enabler for high computational efficiency, since it allows propagating data from unit to unit without needing to store intermediate results for subsequent processing.

3.4. MACROS

Macros can be seen as a set of d-instruction combining parallel and pipelined processing. Macros directly translate into block diagrams, as the one represented in Fig. 2.

The corresponding assembly code (b) clearly shows the data dependencies, and both parallelism and pipelining level among units. Timing diagrams of Fig. 3 show the consistent advantage when using units pipelining.

3.5. PIPELINE BUILDER AND MEMORY ALIAS

Optimized coarse-grain operations implemented through macros can be in turn pipelined by using the pipeline builder. It consists of a set of b-instructions specifically designed to further optimize the execution. Using those instructions the programmer marks each macro as part of a pipeline and sets the rules for data exchange between the pipeline stages.

Fig. 4(a) shows a 3-stage pipeline implemented using the pipeline builder. Since pipeline stages are typically decoupled by memories, additional instructions are provided to implement ping-pong mechanism among those memories. These instructions allow defining a register as a memory alias, i.e. a placeholder for any memory of the d-memory bank. Once the memory alias register has been defined, its content can be toggled so as to implement ping-pong mechanism according to the stage of the processing. Fig. 4 (b) shows how memory aliasing can be used to overcome conflicts on shared memories.

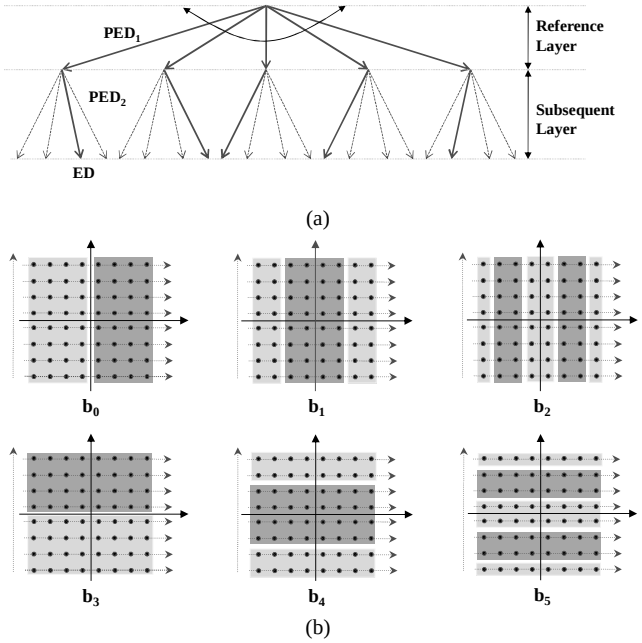


Fig. 5. MIMO tree traversing according to LORD (a) and 64-QAM constellation partitioning for LLR computation (b).

4. MAPPING THE ALGORITHM ON THE BPE

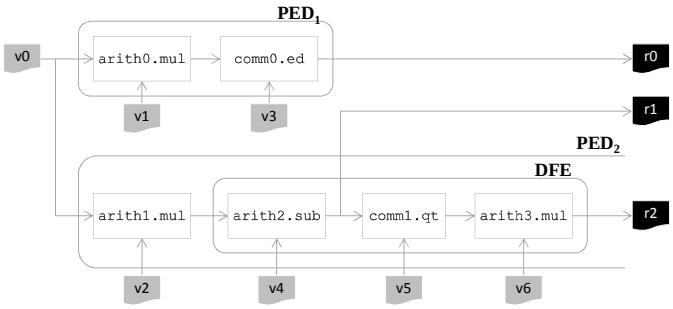
Mapping consists of breaking the algorithm into computing elements and then re-grouping them in a set of macros so as to build a pipeline.

From the system perspective, LORD algorithm basically performs two operations: constellation sweeping and soft-output generation.

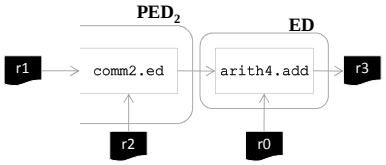
Constellation sweeping consists of computing M^2 EDs per antenna to demodulate M^2-QAM symbols [7]. This can be graphically seen as the tree traversal of Fig. 5(a), showing the computation of the EDs for a 2×R transmission scheme (T = 2). Each ED is the result of the summation of T Partial Euclidean Distances (PEDs), where the PED is defined as the sum of two independent squares related to the I and Q components of a given complex symbol.

For each candidate complex symbol of the reference layer, constellation sweeping involves the operations hereafter summarized:

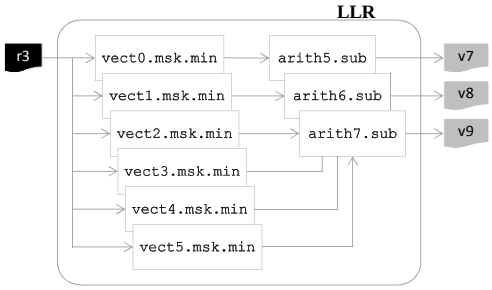
1. compute the PED metrics of the reference layer;
2. compute the PED metrics of each subsequent layer based on spatial DFE;
3. compute the ED metrics as sum of all the PEDs.



(a)



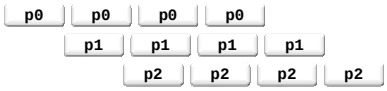
(b)



(c)

```
pipe.def(test)
pipe.set(p0)
pipe.set(p1)
pipe.set(p2)

r0 = malias()
r1 = malias()
r2 = malias()
r3 = malias()

// Gray mapping
v0 = comm0.gray(mod_order)

p0: // macro p0 implementation
    carrier_cnt = add(carrier_cnt,1)
    test = cmp.eq(carrier_cnt,`NB_CARRIER)
    pipe(test)

p1: // macro p1 implementation
    r0 = mtoggle(v10,v11)
    r1 = mtoggle(v12,v13)
    r2 = mtoggle(v14,v15)
    pipe(test)

p2: // macro p2 implementation
    add(llr_cnt,1)
    r3 = mtoggle(v16,v17)
    pipe(test)
```

(d)



(e)

Fig. 6. LORD 3-stage pipeline: macros block diagrams (a, b and c respectively), assembly code template (d) and pipeline timing diagram (e).

Soft-output generation consists of finding the minimum ED among the EDs obtained from constellation sweeping and then computing the LLRs according to (5). The constellation partition depends on the modulation order and the evaluated bit, as shown in Fig. 5(b) for a 64-QAM modulation (darker areas identify EDs with $\mathbf{X_T}$ having $b_{T,1}$ = 1).

Once parallelism and dependencies among data have been identified, the algorithm has been translated into the 3-stage pipeline described in Fig. 6. Macros from (a) to (c) implement the different stages (from p_0 to p_2) of the pipeline.

More specifically, the stage p_0 implements the computation of the PED for the reference layer (PED_1) and part of the PED for subsequent layer (PED_2), including the DFE processing. At this stage, the two layers are independent and thus the calculation of both PED can be performed in parallel, with evident advantage in terms of throughput.

Stage p_1 mainly compute the ED from the PEDs coming from stage p_0. It must be noted that calculation of PED_2 has been split among the two stages p_0 and p_1 to reduce the latency of stage p_0.

Stage p_2 collects all the EDs generated by the stage p_0 and p_1 and evaluate the LLR according to (5).

Assembly template implementing the 3-stage pipeline is shown in Fig. 6(d). Pipeline is defined through the pipeline builder instructions. Each stage is delimited by the corresponding label and the **pipe()** instruction. Pipeline stages make use of memory aliasing, implemented through **malias()** and **mtoggle()** instructions. This allows completely filling the pipeline, as shown in Fig. 6(e).

To fully support data parallelism, the BPE has been equipped with a d-unit bank consisting of 18 units chosen among **arith**, **vect** and **comm** types. Both **arith** and **vect** units offer general purpose fine-grain instructions devoted to complex arithmetic and vector manipulation respectively, while **comm** units provide the programmer with a set of dedicated, but still general purpose instructions within the telecommunications domain. Instructions belonging to the **comm** unit allows, among the others, to calculate the ED between complex numbers, to implement a step function (here used for slicing operation within the DFE) and to compute Gray mapping.

The scheme of Fig. 6 implements LORD algorithm for a generic layer, starting from the processed received signal (3) up to the LLR generation (5). It allows computing one ED

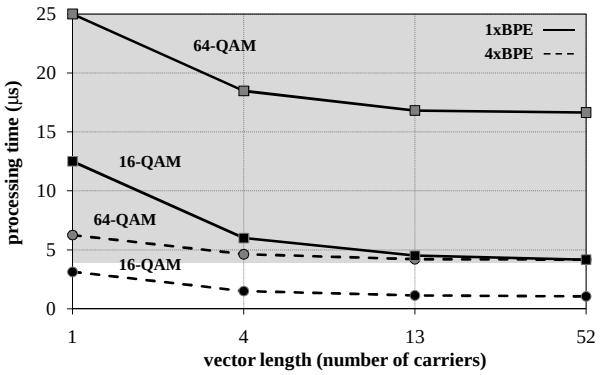


Fig. 7. Processing time with 1 BPE and a cluster of 4 BPEs (dotted) as a function of the vector length (number of carriers).

for each clock cycle, i.e. the same throughput achieved by dedicated hardware architectures, as the one implemented in [10].

The scheme of Fig. 6 is able to process vectors of different lengths. The programmer can choose to process data on an OFDM subcarrier-by-subcarrier basis, or to take advantage of longer vectors by evaluating groups of subcarriers. It must be noted that, especially for low modulation orders, the vector length of one subcarrier (M^2) can result too short to absorb the execution time of the b-instructions controlling the data flow. This ultimately slightly degrades the throughput.

5. RESULTS

Fig. 7 shows the processing time as a function of the vector length (expressed in number of OFDM subcarriers): the values below the gray area satisfy the real-time processing requirements of a MIMO WLAN transmission (i.e, 4 µs) [1]. The plot shows 16-QAM and 64-QAM modulations, for both a single BPE and a cluster of 4 BPEs running at a clock frequency of 350 MHz. The 16-QAM modulation can be real-time processed by a single BPE using a minimum vector length of 13 subcarriers, while for the most demanding 64-QAM modulation a cluster of 4 BPEs is needed.

Lastly, TABLE I summarizes the specification and the implementation results of the above configuration.

6. ACKNOWLEDGEMENTS

The authors would like to thank their colleague M. Siti for his valuable contributions on the LORD algorithm understanding.

TABLE I
BPE SPECIFICATIONS AND IMPLEMENTATION RESULTS
(STMICROELECTRONICS 65NM CMOS TECHNOLOGY)

ARCHITECTURE SPECIFICATION

instructions type	fine-grain
D-unit bank size	18
D-unit bank customization	arith (8) – vect (6) – comm (4)
Register file	32×16 bit
D-memory bank size	18×256×32 bit
I-memory	512×32 bit

IMPLEMENTATION RESULTS

	1 BPE	cluster of 4 BPEs
Clock	350 MHz	
Area	0.9 mm^2	3.9 mm^2
Max Gops (16-bit real ops)	12 Gops	48 Gops
Near-ML detector Gops	9.6 Gops	38.4 Gops
Utilization (%)	80 %	
Throughput (16/64-QAM)	50/40 Mbps	185/150 Mbps

7. REFERENCES

[1] A. Stephens *et. al.*, "Draft amendment to […]-part 11: Wireless lan medium access control (MAC) and physical layer (PHY) specifications: Enhancements for higher throughput," IEEE P802.11nTM/D2.0, 2008.

[2] C. Studer, A. Burg, and H. Bölcskei, "Soft-output sphere decoding: algorithms and VLSI implementation," IEEE Journal on Selected Areas in Communications, vol. 26, no. 2, pp. 290–300, February 2008.

[3] T. Cupaiuolo, M. Siti, and A. Tomasoni, "Low-complexity high throughput VLSI architecture of soft-output ML MIMO detector," in DATE. IEEE, 2010, pp. 1396-1401.

[4] O. Paker, S. Eckert and A. Bury, "A Low Cost Multi-Standard Near-Optimal Soft-Output Sphere Decoder: Algorithm and Architecture," in DATE, 2010, pp. 1402–1407.

[5] B. Bougard, B. De Sutter, S. Rabou, D. Novo, O. Allam, S. Dupont, L. Van der Perre, "A Coarse-Grained Array based Baseband Processor for 100Mbps+ Software Defined Radio", DATE. IEEE, 2008.

[6] S. Eberli, A. Burg, and W. Fichtner, "Implementation of a 2×2 MIMO-OFDM receiver on an application specific processor," Microelectron. J., vol. 40, no. 11, pp. 1642–1649, 2009.

[7] M. Siti and M. Fitz, "A novel soft-output layered orthogonal lattice detector for multiple antenna communications," in IEEE International Conference on Communications, ICC'06, vol. 4, 2006.

[8] M. Li, R. Fasthuber, D. Novo, B. Bougard, L. V. der Perre, and F. Catthoor, "Algorithm-architecture co-design of soft-output ML MIMO detector for parallel application specific instruction set processors." in DATE. IEEE, 2009, pp. 1608–1613.

[9] D. Lo Iacono, J. Zory, E. Messina, N. Piazzese, G. Saia, and A. Bettinelli, "ASIP architecture for multi-standard wireless terminals," in DATE. IEEE, 2006, pp. 118–123.

[10] P. Bhagawat, R. Dash, and G. Choi, "Dynamically reconfigurable soft output MIMO detector," in ICCD, 2008, pp. 68–73.