IP-SOC 2011

Unified C-programmable ASIP architecture for multi-standard Viterbi, Turbo and LDPC decoding

F. Naessens, P. Raghavan, L. Van der Perre, A. Dejonghe
IMEC

Leuven, Belgium

Abstract:

This paper describes an ASIP decoder template suitable for multi-standard Viterbi, Turbo and LDPC decoding. We show architecture fitness for WLAN, WiMAX and 3GPP-LTE standards, although various other standards can also be mapped, since the architecture is capable of supporting any interleaver pattern and programmable in C. The ASIP core consists out of a SIMD with multiple slots each with their dedicated functionality. Because of their block based approach and possible parallelization decoding strategy, both Turbo and LDPC were mapped using the same concept. Support for Viterbi decoding is made possible through a dedicated decoding pipeline with radix-4 to boost performance well above the tough throughput and latency requirements of the 802.11n standard. Specific instantiations are made to show the flexibility of the architecture. For each of these instances, area and throughput is given for a commercial 40nmG technology, showing to be competitive versus other flexible solutions, while offering some key differentiators in the sense of flexibility, usage and specific mode instantiation.

I. INTRODUCTION

Existing and emerging mobile communication networks are posing high requirements and flexibility, which paves the way for Software Defined Radio (SDR). A lot of advances were made in the inner modem processing [1][2][3][4], yielding to architectures that offer a high degree of parallel processing. The outer modem processing, more specifically in the channel decoding, is mainly dominated by dedicated ASIC solutions.

Recently some flexible implementation have been derived which support multi-standard channel decoding [5][6][7]. These solutions usually have a distinct pipelining for each of the decoding modes. With our solution we want to offer multi-standard decoding with maximum hardware re-use and full flexible memory allocation. Our previous solution offered support for flexible Turbo and LDPC decoding [8]. However we have further extended the architecture with extended functional support and enhanced throughputs while having a lower area footprint. Additionally there is a C-compiler with supports mapping functionality. Our solution proves to be flexible (both in supporting multi-standard LDPC, Turbo and LDPC as well as usage) and performant compared to State-of-the-art solutions [5][6][7].

The rest of the paper is structured as follows: Section II introduces the algorithms and how parallelization is exploited. In section III, the architecture is presented together with the instruction set and pipelining. Implementation results and benchmarking are detailed in sections IV and V respectively. Finally, section VI will conclude the paper.

II. DECODING ALGORITHM OVERVIEW

In contrary towards Viterbi decoding, low-density-parity-check (LDPC) and turbo codes are performing very close towards the Shannon limit. The main drawback however is their high computational complexity, which paves the way for parallel decoding architectures. In the next subsections all decoding modes will be detailed and specific focus will be given towards parallel decoding possibilities.

A. *Viterbi decoding*

Convolutional encoding is mandatory present in many communication standards. In a convolutional encoder, data bits are being fed into delay line, from which certain branches are xor-ed and fed to the output. If we consider WLAN as example, the throughput is being stressed towards decoder output rates of 600 Mbps (in IEEE 802.11n standard). Viterbi decoding is a near-optimal decoding of convolutional encoded data. It has a strongly reduced complexity and memory requirement compared to optimal decoding. It was introduced by Andrew J. Viterbi in his landmark paper [9].

During decoding, the most probably path is being determined based on the (soft) information bits and possible delay line state. Specifically in Viterbi, a window (with so called trace-back length) is considered before taking a decision on the most probable path and corresponding decoded bit. A high-level view on the decoding operation is depicted on Figure 1.

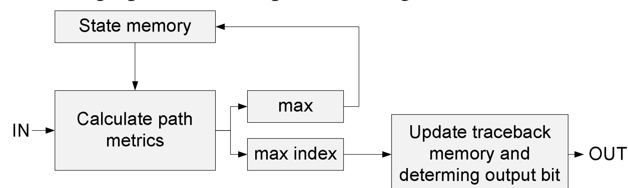


Figure 1 : Decoding overview

There is an iterative path regarding calculation of the state memory, which can be broken into parallel execution by applying a radix-2Z also called look-ahead factor Z [10]. In the remaining of this paper we will focus on a Viterbi decoder suited for WLAN operation, with polynomials equal to 133oct, 171oct. There is fixed decoding pipeline with dedicated registers for which a high level diagram is depicted on Figure 2. We have applied a look-ahead of factor 2 (also called radix-4

implementation), which could result in throughput performance equal to 2 output bits per clock cycle.

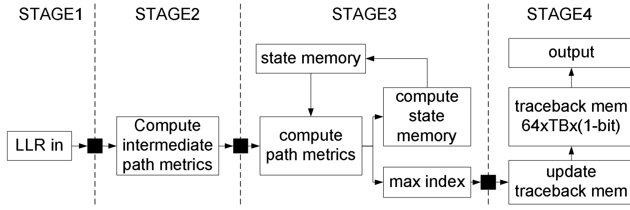


Figure 2 : Viterbi decoding pipeline

B. Turbo decoding

Decoding a turbo code requires iterating maximum a posteriori (MAP) decoding of the constituent codes and (de)interleaving operations. We have chosen to use the log-MAP with max* approximation. The parallelization of the log-MAP can be done either by augmenting the radix factor [11] or by computing in parallel multiple "windows" out of the coded block [12]. Figure 1 depicts the operation of a parallel log-MAP decoder with parallel windows. The top part (a) recalls the classical sliding windows log-MAP decoding flow. A forward recursion, given by (1) is applied to the input symbol metrics (γ). Part of the backward recursion (2) is started after a window of M inputs is treated by the forward recursion. The recursion is then initialized either using a training sequence applied on the symbol metrics from the adjacent window, or using backward state metrics (β) from a previous iteration (next iteration initialization, NII). Based on the state metrics output of the forward (α) and backward (β) recursions, next to the input symbol metrics (γ), the log-MAP outputs (extrinsic information, e) can be computed. Using NII is preferred as it leads to more regular data flow. That way, as shown in the bottom part of the figure (b), the log-MAP decoding can easily be parallelized by computing several windows in parallel.

$$\alpha'_k(m) = \max*_{m'}(\alpha'_{k-1}(m') + \gamma'_k(m',m)) \quad (1)$$

$$\beta'_k(m) = \max*_{m'}(\beta'_{k+1}(m') + \gamma'_{k+1}(m',m)) \quad (2)$$

$$e(d_k) = \max*_{m,m'}(\alpha'_{k-1}(m') + \gamma^1{}_k(m,m') + \beta'_k(m)) - \max*_{m,m'}(\alpha'_{k-1}(m') + \gamma^0{}_k(m,m') + \beta'_k(m)) - L_{syst} - L_{apriori}$$

(3)

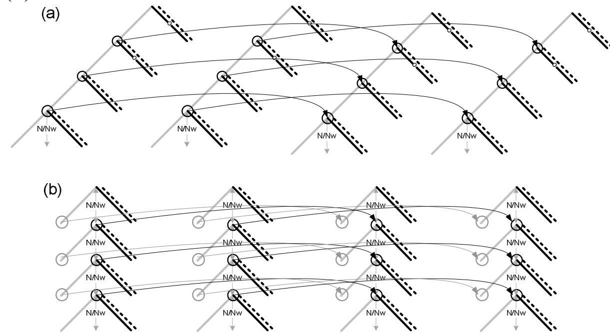


Figure 1 : Parallel log-MAP decoder

Besides the log-MAP, interleaving operations must also be parallelized. As interleaving means intrinsically permutation the data out of a memory, parallelizing it requires to use multiple memory banks and handle potential collision in read and write operations. For this, collision free interleaving patterns or collision-resolution architectures were proposed in [13] and [14] respectively. Later, Tarable proved in [15] that any interleaving law can be mapped onto any parallel multi-banked architecture. He also proposed an annealing algorithm that is proven to always converge to a valid mapping function.

In this work, we are capitalizing on the multi-window log-MAP parallelization combined with Tarable's interleaving law mapping. Contrarily as what we did in [12] however, we do not parallelize the forward and backward recursion anymore. This is to enable single instruction multiple data (SIMD) implementation of the datapath. If SIMD is used to parallelize multiple windows, a restriction is that the same operation has to be applied to all the parallelized windows at the same time. With our original parallelization [12], each "worker" (structure implementing two adjacent windows) had to operate forward and backward steps in parallel, which is impossible in SIMD. Also, for the same reason, the computation of γs, αs and βs over the branches or states are serialized.

C. LDPC decoding

When considering a (n,k) LDPC code, with k information bits and n coded bits the parity check matrix H is of dimension [(n-k)×n]. Simulations have shown that the layered decoding reduces the number of iterations needed to converge [16][17]. In layered decoding [18] the parity check matrix needs to be decomposable into m sub-matrices with size [z×n], also called super-codes as depicted on Figure 2. For every super-code, the position of 1s is restricted such that, per column in a super-code, only zero or one non-zero element is allowed. For the standards in scope this property is either automatically met, through the fact that the sub-matrices within the super-codes are either zero matrices or cyclic permuted identity matrices, or can be derived.

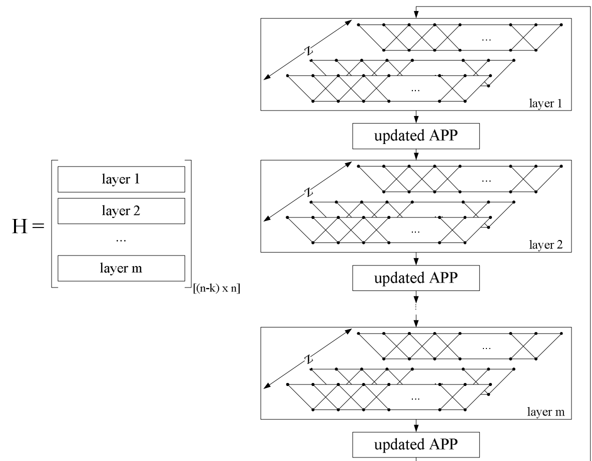


Figure 2 : LDPC Layered Decoding

Due to the latter property, within a layer, the check node updates and corresponding bit node updates can be done in parallel since there is only one edge from every bit node towards a check node in a layer. Intrinsically this enables z parallel processing threads each performing check node updates and message node updates corresponding to a row within the layer of the parity

check matrix. An overview of the possible z values within the standards in scope is summarized in Table 1. In order to achieve the best BER performance, the turbo-decoder-based approach, with the max* [16] approximation is favored. Within the z parallel processing threads, the calculation of the forward and backward recursion metrics as well as the bit node update messages can be calculated.

Table 1 : z-value Within Standards in Scope

Standard	*z value*
IEEE802.11n	{27,54,81}
IEEE802.16(e)	{24,28,32, … , 92,96}
DVB-S2/T2	360

III. ARCHITECTURE OVERVIEW

Based on the parallelization intrinsically present in the algorithms detailed in section II, a SIMD implementation covers the constraints. An overview of the architecture is depicted on Figure 3.

The architecture exists out of a SIMD engine connected to scalar data memory and two (local) vector memories. In order to comply with all permutation and interleaving patterns, the background memory is connected through a crossbar. Both the address and the crossbar control are configured through lookup table which is driving by a virtual address.

The SIMD engine was developed using Target tool suite [19], offering next to high level modeling of processor resources also a C-compiler to efficiently explore and map the decoding applications. There are six parallel slots for which the first one is operating on scalar field only and mainly used for initialization and loop control. The processor has following properties:

- instruction width of 80 bits
- scalar word width of 12 bits (signed)
- vector word width of 8 bits (signed)
- 16 scalar registers
- 8 vector registers
- Maximum pipeline depth of 8 (only utilized for loads from background vector memory)
- Stack implemented on DM and VMEM memory

The instruction set is depicted on Table 2. The instruction set is quite limited and still offers support for most of the C-code constructs.

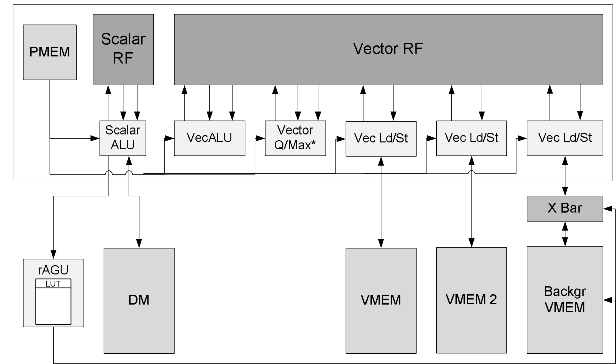


Figure 3 : Architecture overview

Table 2 : Instruction set overview

Slot	*Group*	*Instruction*	*Description*	*semantic*
scalar	movi	movi	Loading constant	_r_dest = _imm_12
	mov	mov	Register move	_r_dest = _r_src
	salu	add_imm	Adding immediate (signed) to register	_r_dest = _r_src + _imm_5
		right shift	Arithmetic right shift	_r_dest = _r_src >> _imm_4
		mult_imm	Multiplying with unsigned immediate	_r_dest = _r_src * _imm_4
		or, and	Logical or/and	_r_dest = _r_src1 {|,&} _r_src2
		add, sub	Arithmetic add, sub	_r_dest = _r_src1 {+,-} _r_src2
	compare	{<,<=,>,>=,==,!=}	Logical comparison (signed)	_SREG = _r_src1 {<,<=,>,>=,==,!=} _r_scr2
	jumps	{call, ujmp, cjmp}	Call, un- and conditional jumps	Call, Ujump _addr, Cjump _SREG _adr
	ret	ret	Return from jump	Ret
	DM_access	DM Load/store	Loading/storing from data memory with optional post-increment	_r_dest = DM[_r_addr {++}] DM[_r_addr {++}] = _r_src
	SP_access	DM Load/store	Loading/storing from data memory through stack pointer with offset	_r_dest = DM[_SP + _imm_7 DM[_SP + _imm_7] = _r_src
Vector basic	valu	{vadd, vsub }	Vector add/subtract	_vr_dest = _vr_src1 {+,-} _vr_src2
	vmani	{vvsl, vvsr}	Vector shift left/right	_vr_dest = _vr_dest {<<,>>}
		vedit	Set specific slot to a specific value	_vr_dest[_r_addr] = _r_value
		vset	Initialize vector with specific value	_vr_dest[0...Z] = _r_src
Vector spec	vq	vq	Elementary LDPC decoding operation	_vr_dest = vq(_vr_src1,_vr_src2)
	vmax	vmax, vmaxnorm	Elementary Turbo decoding operation	_vr_dest = vmax(_vr_src1,_vr_src2{,_vr_norm})
	vvit	vvit	Elementary viterbi decoding (add-compare-select, traceback mem update)	_vr_dest[_r_pos] = vit(_vr_src1[_r_pos],_vr_src2[_r_pos])
Vmem	DMV_access	DMV load/store	Load/store from vmem with post increment (signed)	_r_dest = VMEM[_r_addr {+_imm_4}] VMEM[_r_addr {+_imm_4}] = _r_src
	SPV_access	DMV load/store	Load/store from vmem through stack pointer with offset	_vr_dest = DM[_SPV + _imm_7 DM[_SPV + _imm_7] = _vr_src
Vmem 2	DMV2_access	DMV2 load/store	Load/store from vmem with post increment (signed)	_r_dest = VMEM2[_r_addr {+_imm_4}] VMEM2[_r_addr {+_imm_4}] = _r_src
backgr VMEM	DMVB_access	DMVB load/store	Load/store from vmem with post increment/decrement	_r_dest = DVMB[_r_addr {++,--}] DMVB[_r_addr {++,--}] = _r_src

An example of a firmware kernel and it corresponding assembly can be found in Table 3 and Table 4. The code is part of the LDPC decoding which decodes a layer existing out of 6 non-zero parity check sub-matrices. The loops are automatically unrolled (through compiler directives) and mapped onto 29 cycles.

Table 3 : Firmware C-code example

```
vchar chess_storage(DMV2)   upd[100];
vchar chess_storage(DMV2)   alfa[22];
vchar chess_storage(DMV)    tmp[22];
vchar chess_storage(DMVB:0) llrs[256];

void perform_layer6_decoding(int start_idx){
  vchar llr_curr, alfa_curr;
  vchar beta, llr_prev, output;
  alpha = vinit(-32); beta  = vinit(-32);
  int mem_pointer = start_idx;
  // forward
  for(int i=0;i<6;i++){
    llr_curr = llrs[mem_pointer];
    llr_prev = upd[mem_pointer++];
    llr_curr = llr_curr - llr_prev;
    alfa[i] = alfa_curr;
    llr_tmp[i] = llr_curr;
    alfa_curr = vq(alpha,llr_curr);
  }
  // backward + output
  for(int i=5;i>=0;i--){
    alfa_curr = alfa[i];
    output = vq(alfa_curr, beta);
    upd[--mem_pointer] = output;
    llr_curr = tmp[i];
    output = output + llr_curr;
    beta = vq(beta, llr_curr);
    llrs[mem_pointer] = output;
  }
}
```

The slot utilization for the different processing kernels is depicted on Figure 4. One can clearly observe that the viterbi decoding kernel doesn't utilize the vector alu and local vector memory slots. In overall, a quite dense schedule is achieved, certainly taking into account the very limited number of vector registers. Increasing the number of vector registers would be beneficial for the schedule, but would drastically increase the occupied area and power consumption of the SIMD processing core.

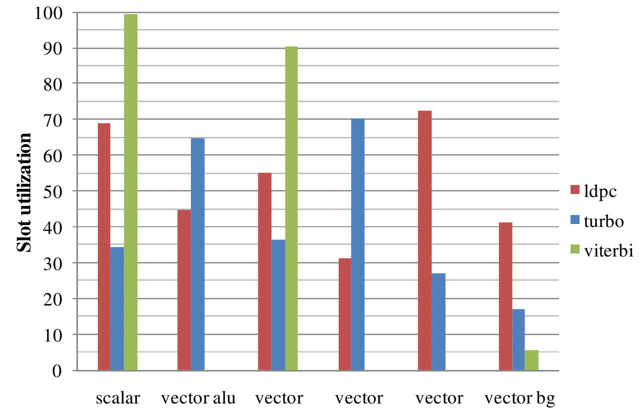


Figure 4 : Slot utilization [%] for the different processing kernels (turbo, ldpc, viterbi)

Table 4 : Resulting assembly code mapped over different ASIP core slots

```
movi R1 upd   || nop              || nop            || nop            || nop            || nop
add R1 R0 R2  || nop              || nop            || nop            || nop            || VR0=DMVB[R0++]
add R1 R0 R3  || nop              || nop            || nop            || VR6=DMV2[R2]   || VR1=DMVB[R0++]
add R1 R0 R2  || nop              || nop            || nop            || VR3=DMV2[R3]   || VR2=DMVB[R0++]
movi R3 -32   || nop              || nop            || nop            || VR5=DMV2[R2]   || nop
add R1 R0 R2  || vset VR4 R3      || nop            || nop            || nop            || VR7=DMVB[R0++]
movi R6 tmp   || nop              || nop            || nop            || nop            || nop
add R1 R0 R4  || vsub VR0 VR6 VR0 || nop            || nop            || VR0=DMV2[R2]   || nop
mov R0 R2     || vsub VR1 VR3 VR1 || vq VR4 VR0 VR3 || DMV[R6+1]=VR0  || nop            || VR1=DMVB[R0++]
movi R8 alfa  || nop              || vq VR3 VR1 VR6 || DMV[R6+1]=VR1  || nop            || VR2=DMVB[R0]
mov R4 R3     || nop              || nop            || nop            || DMV2[R8+1]=VR4 || nop
add R1 R0 R5  || vsub VR2 VR5 VR3 || nop            || nop            || DMV2[R8+1]=VR3 || nop
nop           || vsub VR7 VR0 VR0 || vq VR6 VR3 VR3 || DMV[R6+1]=VR3  || VR5=DMV2[R4]   || nop
mov R5 R4     || nop              || nop            || nop            || VR6=DMV2[R5]   || nop
mov R6 R7     || vsub VR1 VR5 VR1 || vq VR3 VR0 VR0 || DMV[R6+1]=VR0  || DMV2[R8+1]=VR6 || nop
mov R8 R5     || vsub VR2 VR6 VR5 || vq VR0 VR1 VR2 || DMV[R6]=VR1    || DMV2[R8+1]=VR3 || nop
nop           || nop              || vq VR2 VR4 VR6 || VR2=DMV[R7-1]  || DMV2[R8]=VR0   || nop
nop           || vadd VR6 VR5 VR5 || vq VR4 VR5 VR4 || nop            || VR3=DMV2[R5-1] || nop
nop           || nop              || vq VR0 VR4 VR0 || VR0=DMV[R7-1]  || DMV2[R4]=VR6   || DMVB[R0]=VR5
nop           || vadd VR0 VR1 VR1 || vq VR4 VR1 VR4 || VR3=DMV[R7-1]  || DMV2[R3]=VR0   || nop
nop           || nop              || vq VR3 VR4 VR6 || nop            || VR1=DMV2[R5-1] || DMVB[R2--]=VR1
add R1 R2 R0  || vadd VR6 VR2 VR2 || vq VR4 VR2 VR7 || VR1=DMV[R7]    || VR5=DMV2[R5-1] || nop
nop           || nop              || vq VR1 VR7 VR4 || nop            || DMV2[R0]=VR6   || DMVB[R2--]=VR2
add R1 R2 R0  || vadd VR4 VR0 VR6 || vq VR7 VR0 VR0 || nop            || VR2=DMV2[R5]   || nop
nop           || nop              || vq VR5 VR0 VR4 || nop            || DMV2[R0]=VR4   || DMVB[R2--]=VR6
add R1 R2 R0  || vadd VR4 VR3 VR3 || vq VR0 VR3 VR0 || nop            || nop            || nop
ret           || nop              || vq VR2 VR0 VR0 || nop            || DMV2[R0]=VR4   || DMVB[R2--]=VR3
add R1 R2 R0  || vadd VR0 VR1 VR1 || nop            || nop            || nop            || nop
nop           || nop              || nop            || nop            || DMV2[R0]=VR0   || DMVB[R2]=VR1
```

IV. IMPLEMENTATION RESULTS

A. Template instantiation

Based on the requirements of the decoding modes a parallelization of 96 slots is selected. We considered three distinct instances for which the supporting modes are summarized in Table 5. In order to allow a comparison, internal quantization was fixed towards 8 bit for all instantiations.

Table 5 : Distinct instantiations

	WLAN LDPC	WiMAX LDPC	3GGP-LTE Turbo	WLAN Vit
LDPC	X	X		
LDPC/Turbo	X	X	X	
LDPC/Turbo/Vit	X	X	X	X

The mapping onto the memory hierarchy for each of the decoding modes is depicted on Figure 5. Since the Viterbi decoding has a dedicated pipeline, there is no need for usage of the local vector memory. The memory requirements are clearly driven by the Turbo decoding mode.

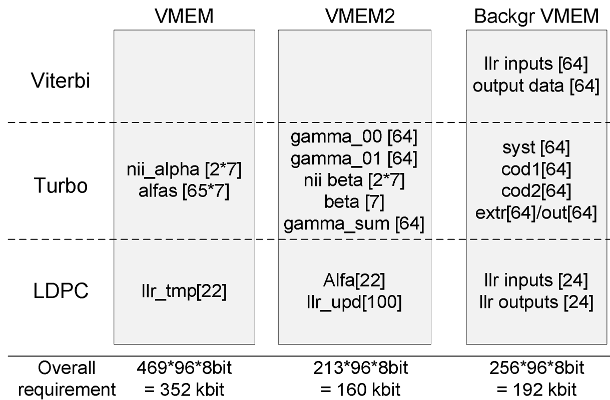


Figure 5 : Variable mapping onto memory hierarchy for different deocoding modes

An overview of the implementation details for each of the instances is given in Table 6. In case of the LDPC only instance, considerable amount of complexity can be avoided. First the number of scalar and vector registers can be reduced significantly without sacrificing throughput performance; secondly the full crossbar can be replaced by a rotation engine (barrel shifter) due to the properties inside the WLAN and WiMAX LDPC codes. On top of that the memory requirements are much lower and the background memory doesn't need to be segmented separately over the 96-slots.

Table 6 : implementation details for different instances

	LDPC	LDPC/Turbo	LDPC/Turbo/Vit
Number of vector registers	5	8	8
Number of scalar registers	10	16	16
Shuffler	barrel shifter	full crossbar	full crossbar
VMEM size	22x(96x8b)	469x(96x8b)	469x(96x8bit)
VMEM2 size	122x(96x8b)	213x(96x8b)	213x(96x8bit)
backgr VMEM size	24x(96x8b)	(4*64)x96x8b	(4*64)x96x8bit

B. Area

For the implementation, a 0.9V 40nm process was selected. Logic synthesis was performed using commercial tools and a floorplan utilization of 60% was taken into account. Memory area is based on macros of commercial available memory IP vendor. In Figure 6 you can find the area for the ASIP core for each of the template instantiations, which clearly shows the high area impact of the vector registers and the overhead of Viterbi decoding as completely independent pipeline (contribution both in 'alu vector spec' as in 'other regs'). The overall area for each of the instances can be found in Table 7, which clearly shows the architecture is memory dominated and for this memory constraints, the 3PP-LTE Turbo decoding mode is the most demanding one.

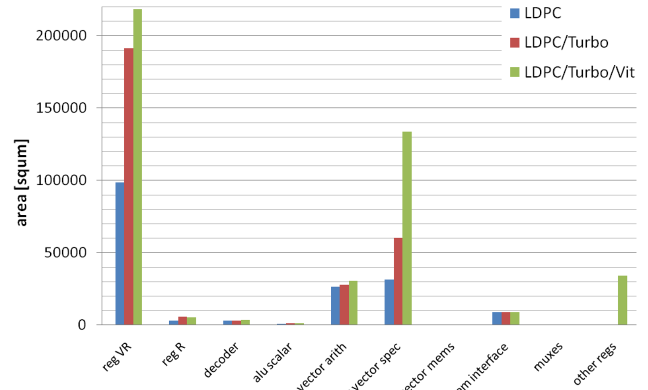


Figure 6 : ASIP area distribution for different instances

Table 7 : Overall area overview for different instances

AREA [squm]		LDPC	LDPC/Turbo	LDPC/Turbo/Vit
LOGIC	ASIP core	172,507	299,863	436,761
	Shuffler	64,035	123,125	123,125
MEMORIES		167,906	688,314	688,314
TOTAL		404,448	1,111,302	1,248,200

C. Throughput

An overview of some of the throughputs which can be achieved are shown in Table 8. As for the Viterbi decoding, the throughput is approaching a two output bits per clock cycle. The overhead is coming from the memory access and loop control

Table 8 : Throughput results for 800MHz clock frequency

Algorithm	Mode	Throughput Mbps single it.
Viterbi	802.11n - 64 bits	550.5
	802.11n - 1024 bits	1262.2
LDPC	802.11n – Z=81 – CR 5/6	3590.0
	802.16e – Z=96 – CR 5/6	4208.2
Turbo	3GPP – 6144 bits – CR 1/3	541.1
	3GPP – 40 bits – CR 1/3	10.8

D. Power/energy efficiency

An overview of the power consumption and according decoding energy efficiency is depicted on Figure 8 and Figure 9 respectively. The leakage is relatively small compared to the dynamic power consumption. The power consumption (hence also energy efficiency) is slightly increasing from the LDPC only instance compared to the full flexible solution.

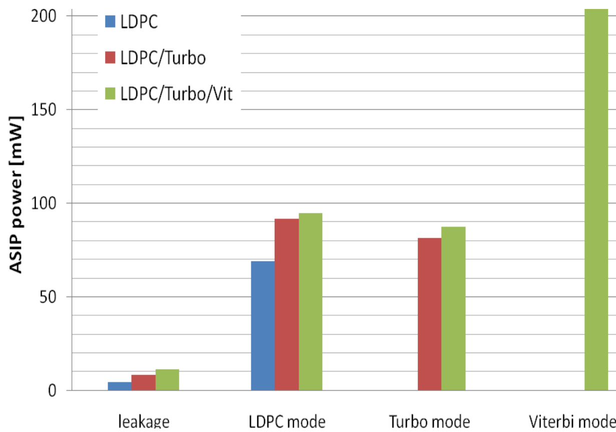


Figure 7 : ASIP power consumption

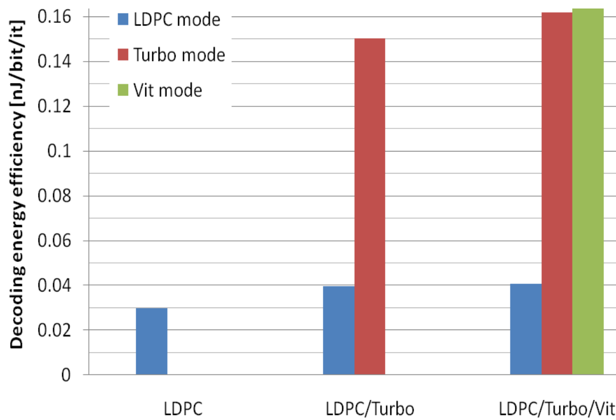
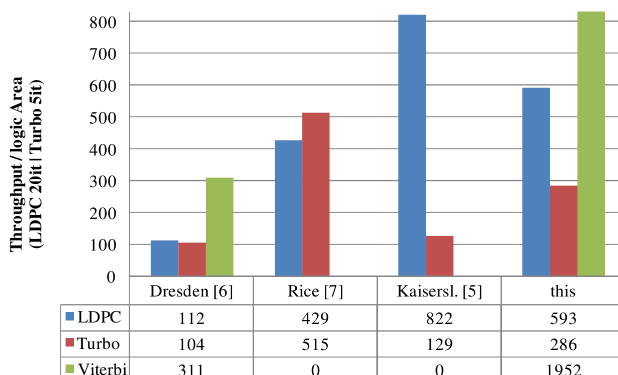


Figure 8 : Decoding efficiency for different modes mapped onto different instances

V. COMPARISON VERSUS STATE-OF-THE-ART

Comparison of our solution versus other flexible solutions is shown on Figure 9. All solutions were compared with a throughput over logic-area metric, were area numbers were scaled towards 40nm technology. Our solution show to be very competitive versus other solutions and further tuning is possible when removing the Viterbi and/or Turbo decoding mode support. On top of that we offer a high degree of flexibility through support of any interleaver pattern and C-programmable core. In addition, some of the proposed solutions [5][6] will not be able to meet the demanding latency and throughput constraints posed by the WLAN standard.



	Dresden [6]	Rice [7]	Kaisersl. [5]	this
LDPC	112	429	822	593
Turbo	104	515	129	286
Viterbi	311	0	0	1952

Figure 9 : Flexible multi-standard comparison (based on throughput/logic-area ratio)

VI. CONCLUSIONS

We have shown a flexible architecture which can be seen as a template suited for multi-standard multi-mode LDPC, Turbo and Viterbi decoding. Both LDPC and Turbo can be nicely parallelized to be mapped onto a SIMD engine. In order to boost the throughput, the SIMD engine consists out of parallel slots, each with their dedicated functionality. Viterbi support is enabled through a dedicated pipeline for which we have selected a radix-4 for enhanced throughput.

Some specific instantiations of the template have been implemented in a commercial 40nmG technology and show to be competitive versus other state-of-the-art solutions, while offering clear benefits in flexibility, usage and specific mode instantiation.

REFERENCES

[1] B. Bougard et al., “Energy Efficient Software Defined Radio Solutions for MIMO-based Broadband Communication”, Proc. European Signal Processing Conference, Sept. 2007”

[2] B. Bougard et al., “A Coarse-Grained Array based Baseband Processor for 100Mbps+ Software Defined Radio”, Design Automation and Test in Europe, March 2008

[3] J. Glossner et al., “The Sandblaster SB3011 platform”, EURASIP Journal on Embedded Systems, 2007

[4] Yuan Lee et al., “SODA: A High-Performance DSP Architecture for Software-Defined Radio”, IEEE Micro, 2007

[5] Alles M, Vogt T, Wehn N. FlexiChaP: A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding. *2008 5th International Symposium on Turbo Codes and Related Topics.*

[6] Kunze S, Matuˇ E, Corp NEC. A ” Multi-User ” approach towards a channel decoder for convolutional, Turbo and LDPC codes, TU Dresden Vodafone Chair Mobile Comm . Systems System IP core research labs. *Architecture*. 2010:390-395.

[7] Sun Y, Cavallaro JR. A Flexible LDPC/Turbo Decoder Architecture. *Journal of Signal Processing Systems.* 2010;(November 2009).

[8] Naessens F, Derudder V, Cappelle H, et al. and decoder for 802 . 11n , 802 . 16e and 3GPP-LTE. *VLSI symposium*. 2010:213-214

[9] Andrew J. Viterbi, ”Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”, IEEE Transactions on Information Theory 13(2):260–269, April 1967

[10] G. Fettweis and H. Meyr, “High-Speed Parallel Viterbi Decoding: Algorithm and VLSI-Architecture“, IEEE Comm. Magazine, May 1991.

[11] M Bickerstaff et al., “A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless”, ISSCC, Feb 2003

[12] B. Bougard et al., “A scalable 8.7 nJ/bit 75.6 Mb/s parallel concatenated convolutional (turbo-) CODEC”, ISSCC, Feb. 2003

[13] Giulietti, A.; van der Perre, L.; Strum, A., "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," Electronics Letters , vol.38, no.5, pp.232-234, 28 Feb 2002

[14] Thul, M.J et al, "A scalable system architecture for high-throughput turbo-decoders," IEEE Workshop on Signal Processing Systems, 2002, pp. 152-158

[15] Tarable, A.; Benedetto, S.; Montorsi, G., "Mapping interleaving laws to parallel turbo and LDPC decoder architectures", IEEE Transactions on Information Theory, vol.50, no.9, pp. 2002-2009, Sept. 2004

[16] M. Mansour and N. Shabhag, “High-throughput LDPC decoders”, IEEE Trans. VLSI Syst., 11(6): pages 976-996, Dec 2003

[17] R. Priewasser, M. Huemer, B. Bougard, “Trade-off analysis of decoding algorithms and architectures for multi-standard LDPC decoder”, IEEE Workshop on Signal Processing Systems, 2008.

[18] D. Hocevar, “A Reduced Complexity Decoder Architecture via Layered Decoding of LDPC Codes”, IEEE Workshop on Signal Processing Systems, 2004, pages 107-112.

[19] Target Compiler Technologies, http://www.retarget.com/