AES-OTR v2

Kazuhiko Minematsu (NEC Corporation)
Joint work with Maki Shigeri and Hiroyasu Kubo
(NEC Solution Innovator)

DIAC 2015, 28 September 2015, Singapore

AES-OTR (v2)

- OTR blockcipher mode of operation (Eurocrypt 2014), defined with AES
- Nonce-based AE (NAE)

Features:

- Rate-1 (needs one AES call for one block)
- Parallelizable for encryption/decryption
- Inverse-free

- Unique AES-based NAE candidate achieving all of them

Limitations

- NAE: nonce must be unique
  - Security guarantee is the same as current standards (CCM/GCM)
  - Assumption : AES = PRP (or PRF)
- No protection against nonce-misuse and decryption-misuse

- Theoretical limit speed and size : AES in counter mode

A change from v1

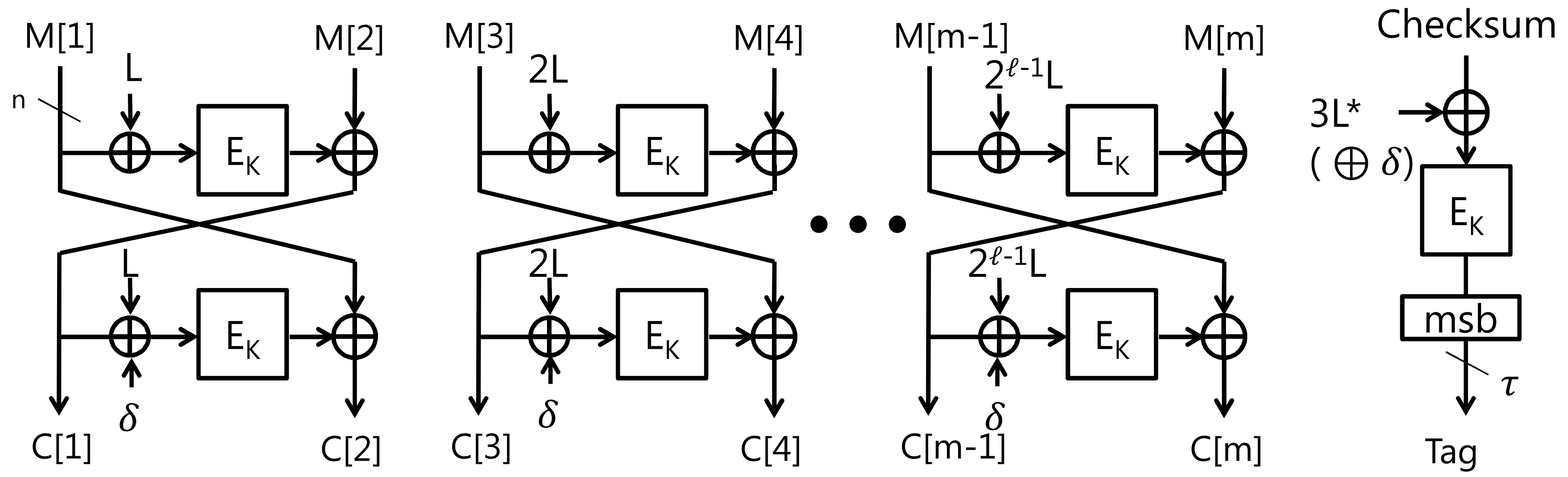
- A small change in nonce processing:
- Tag length (τ) is encoded with nonce encryption
- Format(N, τ) = $(\text{len}(\tau) \bmod n)_7 || 00..01 || N$
  - The same function as OCB
  - Not allowing variable-tag-length (τ is a parameter fixed in use)
  - For preventing trivial misuse (e.g. chopping tag more than what agreed)
  - No performance penalty

Processing of OTR

- Use two-round Feistel permutation with XE-mode as round function
  - Using GF(2^{128}) doubling
- Two-round Feistel can work as double-block cipher in OCB-like AE
  - Checksum is the sum of even blocks
- Last one or two blocks are processed a variant of two-round or CTR to avoid expansion
- Two AD processing versions (PMAC/CMAC) with different points to add MAC output

Basic structure of OTR (no AD)

- δ = E(Format(N, τ))
- L = 4 δ
  - See v2 doc for details



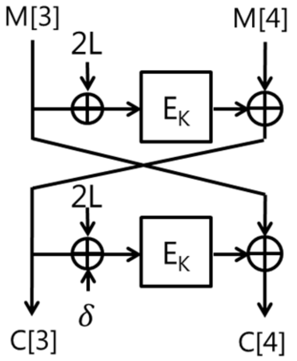
Software performance analysis : the case of AESNI

AES-OTR in AESNI

- AESNI: AES hardware instruction available in modern high-end CPUs (Intel, AMD)
- Basic expectation: AES-OTR runs fast, but slightly slower than AES-OCB
  - According to Bogdanov-Lauridsen-Tischhauser (ePrint 2014, FSE 2015) : the difference in peak speed (i.e. for long messages) : 0.15~0.2 cycles/byte
- A natural tread-off
  - inverse-freeness can be useful for many other platforms (e.g. hardware, embedded CPUs)
  - but closing the gap is good anyway

AES-OTR in AESNI

- One possible reason: GF maskings
- 1st Feistel round: GF doubling
- 2nd Feistel round: additional XOR of δ
    - Which can be absorbed into the first round key, by precomputing $RK_1 \oplus \delta$
- OCB (of CAESAR candidate) does not need doubling in motion ; gray code with precomputed values
    - At the cost with increased memory



GF doubling

- $X \rightarrow (X \ll 1) \oplus \mathrm{msb}(X) \cdot (0x87)$
  - we also need endian conversion, which can be done in one instruction (pshfub)
- We usually perform doubling over 128-bit XMM register
- But there is no single instruction to perform one-bit shift of XMM
- A common method needs 5 instructions
  - OTR uses a doubling for each 32 bytes -> overhead is around 0.15 c/b
    - Can be a major factor
- We review known doubling methods

Krovetz

- (in optimized OCB C code*1)
- 5 instructions
  - A clever way to compute carry by arithmetic right shift (mm_srai_epi32)
  - All fast instructions

Example:

```
void doubling(__m128i in, __m128i *out) {
                const __m128i mask = _mm_set_epi32(135, 1, 1, 1);
                __m128i tmp = _mm_srai_epi32(in, 31);
                tmp = _mm_and_si128(tmp, mask);
                tmp = _mm_shuffle_epi32(tmp, _MM_SHUFFLE(2, 1, 0, 3));
                *out = _mm_slli_epi32(in, 1);
                *out = _mm_xor_si128(*out, tmp);
}
```

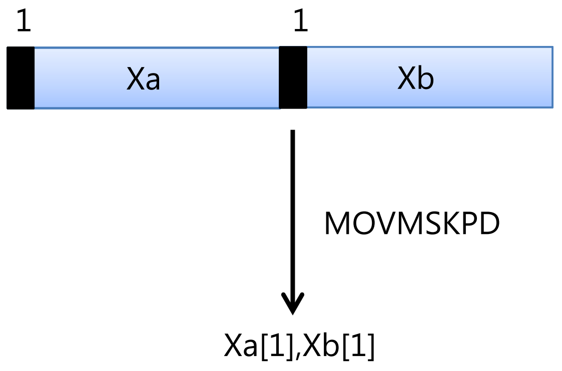
*1 available from (http://www.cs.ucdavis.edu/~rogaway/ocb/)

Aoki-Iwata-Yasuda (DIAC 2012)

- Use movemask (MOVMSKPD) instruction for extracting two MSBs in two 64 bits
- 3 instructions + 1 table lookup
- Table lookup can be costly, MOVMSKPD is not the fastest (latency 3 @ Haswell)

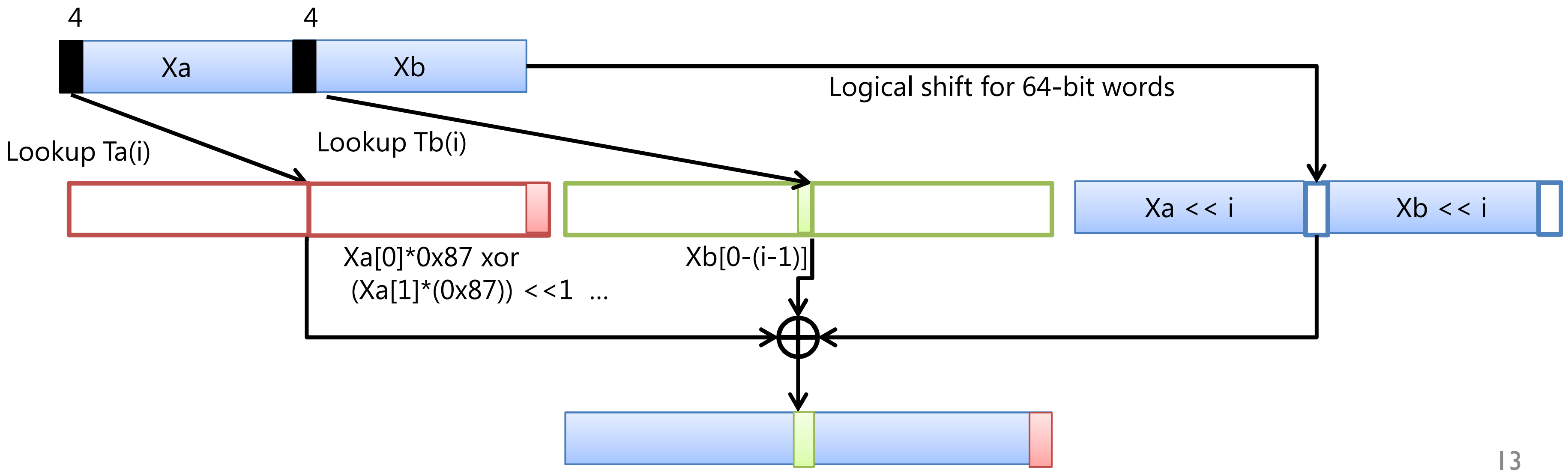
Example:

```
void doubling_aoki(__m128i in, __m128i *out){
                unsigned bm;
                __m128i tbl[4];
                tbl[0]= _mm_set_epi32(0x00,0x00,0x00,0x00);
                tbl[1]= _mm_set_epi32(0x00,0x01,0x00,0x00);
                tbl[2]= _mm_set_epi32(0x00,0x00,0x00,0x87);
                tbl[3]= _mm_set_epi32(0x00,0x01,0x00,0x87);
                bm = _mm_movemask_pd((__m128d)in);
                *out = _mm_add_epi64(in, in);
                *out = _mm_xor_si128(*out, tbl[bm]);
}
```



Aoki (SCIS 2013 (in Japanese))

- Batch doubling : X -> (2X, 4X, 8X,…)
  - Useful for many schemes, incl. OTR
- For X = (Xa, Xb) (each 64 bits),
- 8 16-entry 16-byte tables storing Xa's carry (shift-and-xor of 0x87) and Xb's carry
  - Table lookups for Xa[0-3], Xa[4-7], Xb[0-3], Xb[4-7]
  - Total 2K bytes
- After address calculation, each doubling can be done with 3 fast instructions with 2 table lookups



4
4
Xa
Xb
Logical shift for 64-bit words
Lookup Ta(i)
Lookup Tb(i)
Xa << i
Xb << i
Xa[0]*0x87 xor (Xa[1]*(0x87)) <<1 …
Xb[0-(i-1)]

Other methods

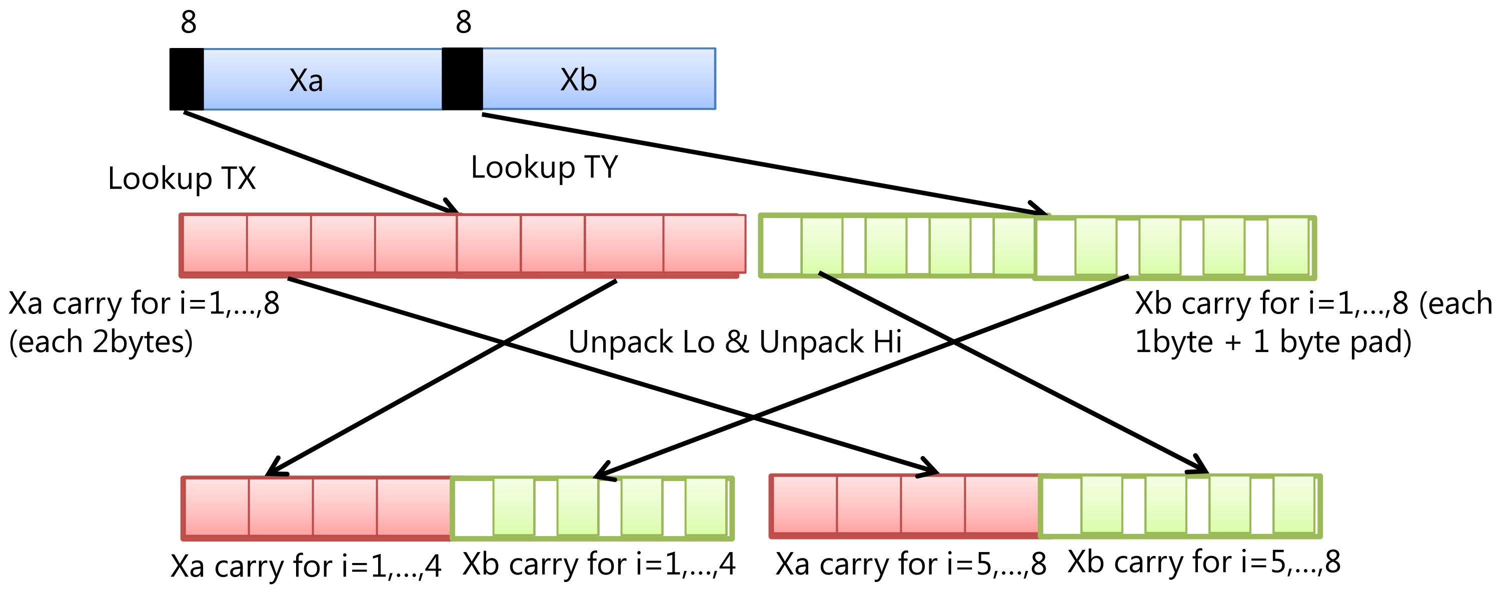
- Bogdanov et al. (eprint 2014, FSE 2015)
  - Similar to Krovetz
  - Investigated various options for implementing "if(MSB(X)==1)"
  - 4 instructions + "if(MSB(X)==1)"
- Chakraborty-Sarkar (ePrint 2014)
  - Survey of known methods and comparison of doublings of different fields
- Our first try
  - (64-bit x 2) version of Krovetz
  - Replace _mm_srai_epi32 + _mm_and_si128 with _mm_maskload_epi64 (VPMASKMOVD in *AVX2*)
  - 4 instructions, but maskload is slow (latency 4, reciprocal t'put 2 @ Haswell), according to Agner Fog

A quest for fast doubling

- We implemented Aoki, Batch Aoki, and Bogdanov et al.
- In our experiment, none of them constantly beat the original Krovetz
  - Except Aoki for 8 batch doublings, which runs almost as fast as Krovetz, and for some cases, it is slightly faster
  - Gain depends on microarchitecture and compiler

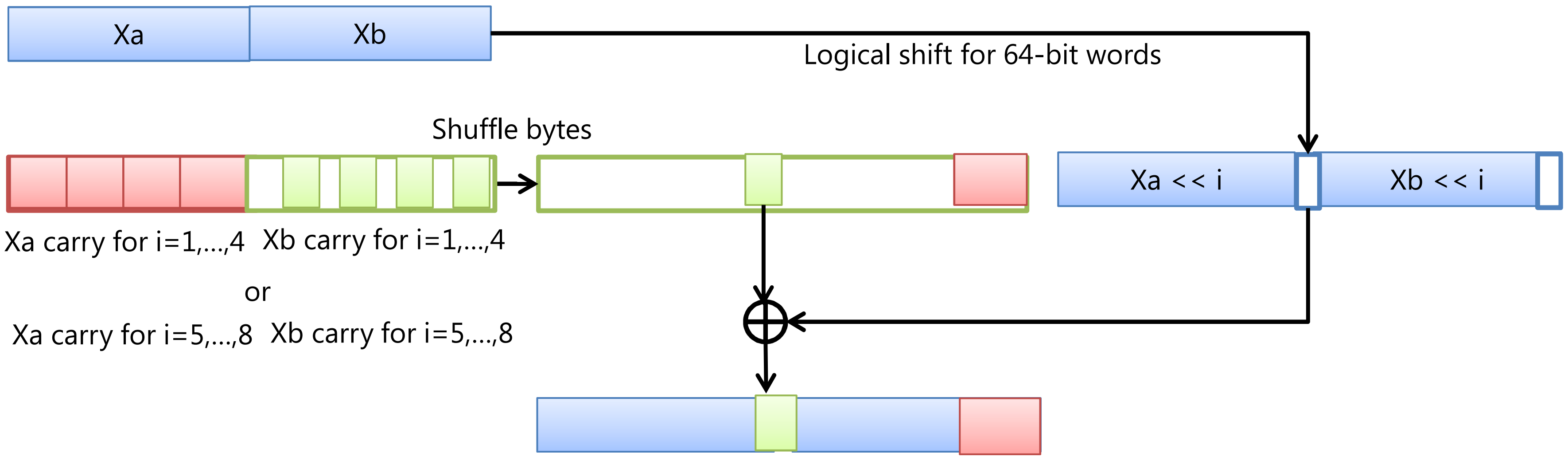
A new batch method for 8 doublings

- Take xmm X as two 64 bits, (Xa, Xb)
- Use two 256-entry 16-byte tables (TX, TY) storing the results of Xa carry (2 bytes) and Xb carry (1 byte + 1 byte pad) for *i*=1,…,8-times doublings
  - total 8K bytes
- Swap upper and lower halves of TX and TY 2 using 2 unpack instructions
- Two table lookups and two unpacks



A new batch method

- Then for each *i*, a dougling (2^i X) can be done by a shuffle instruction (to obtain the *i*-th offset), a shift instruction, and a XOR instruction
- Thus 3 fast instructions



Effect

- We observed improvement from Krovetz
- For example on Haswell Core i7 with gcc 5.1.0
  - (a batch version of) Krovetz and batch Aoki : 4.8
  - The new method : 4.1
  - If with endian conversions, the gain is around 0.5 ~ 1.3 c/b, depending on microarchitecture and compiler
- The same routine can be useful for other CAESAR candidates, such as
  - COPA, ELmD, POET (in MAC part), SHELL
  - Non-CAESAR : PMAC, XTS etc.
- We need care when side-channel analysis against this table lookup is a practical risk

AES-OTR in AES-NI

- The code is written with intrinsic and new batch doubling
- ~10% improvement from [BLT15] @ 2Kbyte message

Haswell Core i7-4770 CPU @ 3.40GHz (CentOS gcc 5.1.0),
AES-OTR w/ AES-128, no AD

Msglen (byte)	Enc (median, c/b)	Dec (median, c/b)
1024	1.02	1.11
2048	0.84	0.86
4096	0.78	0.79

AES-OTR in ARMv7

AES-OTR in ARMv7

- Use Bitslice AES available from SUPERCOP (originally Kasper-Schwabe ,CHES 2009)
  - Done 8 blocks in parallel
- Also single-block AES (standard T-table)
  - Nonce/tag encryption
- Use NEON SIMD engine, intrinsic
- No optimization wrt doubling
- Platform: Beaglebone Black (Cortex-A8 1GHz), with gcc 4.7.3

Results

- Peak speed : ~23.5 c/b (+7% of AESBS)
- For reference, in Gouvea-Lopez (CTRSA 2015) GCM runs 32.8 c/b, using BS-AES, on Cortex A9
- Wanted : fast constant-time single-block AES on ARMv7 (vector-permutation-based one) for gaining performance for short messages

AES-OTR (AES-128, no AD)

Msglen (byte)	Enc (median c/b)	AESBS ratio	Dec (median c/b)	AESBS ratio
1056	25.42	1.14	25.42	1.14
2080	24.19	1.07	24.2	1.07
16416	23.5	1.07	23.51	1.07

AES Bit-slice

Msglen (byte)	Enc (med c/b)
1056	22.24
2080	22.58
16416	21.87

AES T-table

Msglen (byte)	Enc (med c/b)
1056	41.31
2080	43.56
16416	43.54

AES-OTR in FPGA (preliminary result)

AES-OTR in FPGA

- A basic comparison of OTR/OCB/GCM
  - Reference versions (full-spec, unoptimized)
  - All modes use the same AES core (round-based)
  - Implement OTR and OCB. GCM mode taken from OpenCore (no decryption routine…)
  - Additionally, (experimental) dual-core implementation for OTR
- Environments: Altera Cyclone IV (EP4CE30F29C6), Quartus 13.1.4
- Many thanks to Tomonori Iida for implementation

Results

Size

		Logic Cell (AES)	LUT	Register
OTR parallel	ENC	6,647 (3,138)	6,618	1,576
	DEC	7,069 (3,120)	6,927	1,845
	ENC/DEC	8,056 (3,141)	7,901	1,846
OTR parallel AES2 Core	ENC	12,682 (5,483)	12,636	2,392
	DEC	13,041 (5,587)	13,007	2,391
	ENC/DEC	15,773 (5,633)	15,736	2,394
OTR serial	ENC	6,322 (3,144)	6,282	1,448
	DEC	6,814 (3,103)	6,693	1,717
	ENC/DEC	7,631 (3,122)	7,496	1,718

		Logic Cell	LUT	Register
OCB	ENC	7,447 (3,141)	7,357	1,522
	DEC	10,657 (3,053/3,100)	10,657	1,917
	ENC/DEC	11,711 (3,123/3,144)	11,515	1,923

		Logic Cell	LUT	Register
GCM	ENC	7,167 (3,127)	6,973	1,434

Speed

For 12-byte Nonce, 16-byte AD, 32-byte Plaintext

		Clock cycles	Max Freq. (MHz)	Time (ns)
OTR parallel	ENC	72	106.58	676
	DEC	72	108.95	661
	ENC/DEC	72	108.31	665
OTR parallel AES 2 Core	ENC	39	103.63	376
	DEC	49	108.98	450
	ENC/DEC	39/49	96.59	----
OTR serial	ENC	72	105.30	684
	DEC	72	105.83	680
	ENC/DEC	72	104.99	----

		Clock cycles	Max Freq. (MHz)	Time (ns)
OCB	ENC	72	110.98	649
	DEC	72	99.45	724
	ENC/DEC	72	97.90	735

		Clock cycles	Max Freq. (MHz)	Time (ns)
GCM	ENC	80	104.18	768

Rough summary

- OTR is the smallest (in particular, serial ADP is small)
- OTR & OCB run with the same cycles, have similar speed
- If AD is always empty, -1,000 LE is possible

		Logic Cell	LUT	Register
OTR serial, AD empty	ENC	5,640(3,121)	5,601	1,444

- OTR's gain from GCM is moderate
  - As expected: OTR focus the balanced performance on Sw/Hw, from low-end to high-end
- Need more studies…

Thank you!