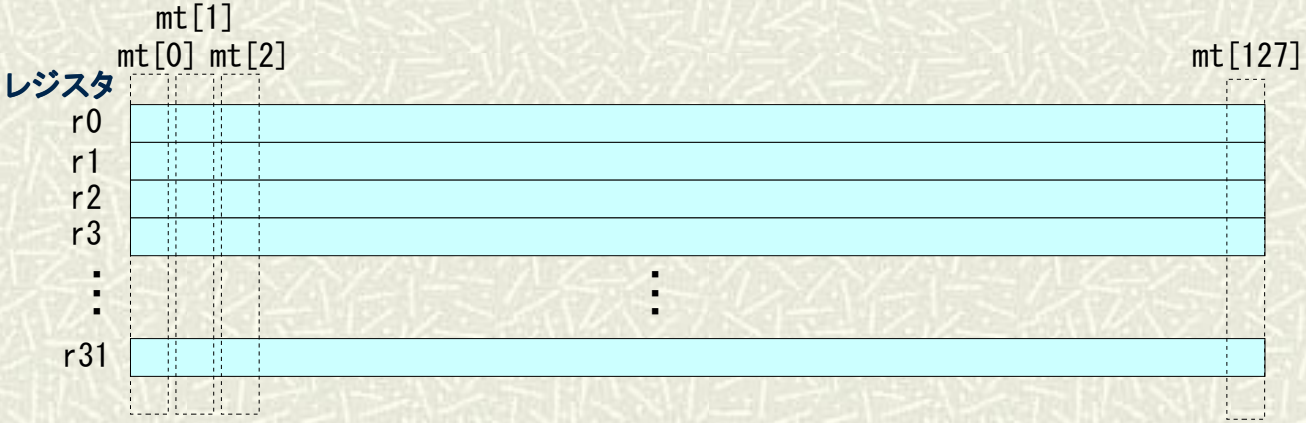
Hack The Cell 2009 を振り返って

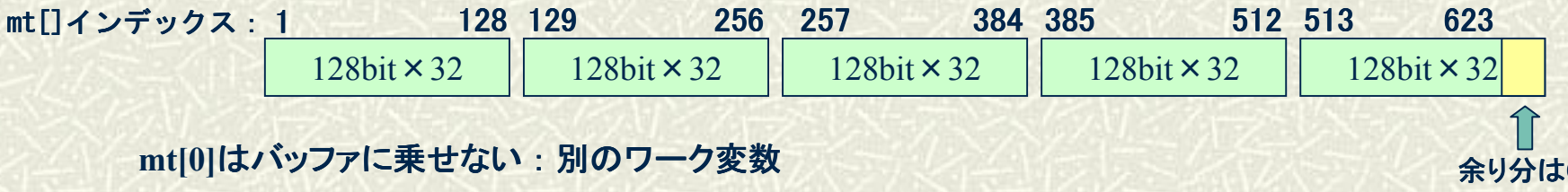
bitslice

▼ mt転置



shuffle + gb + shift の組み合わせで生成。割と重い処理。

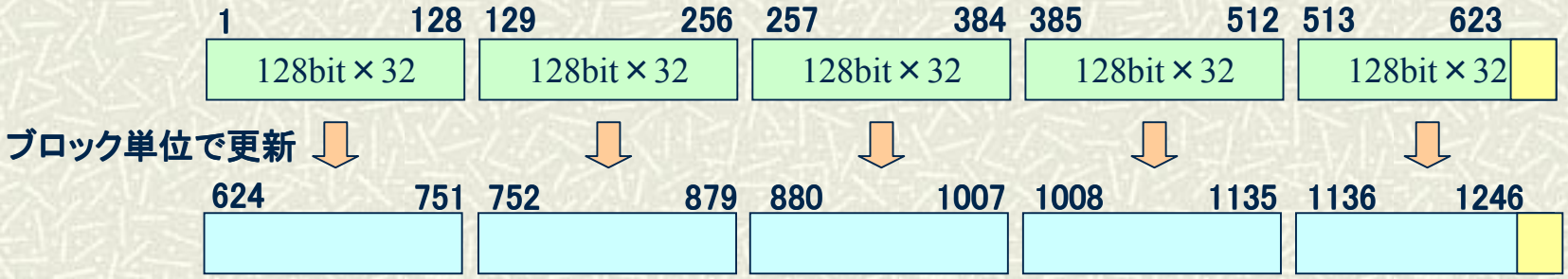
▼ mt[624] の配置



mt[0]はバッファに乗せない：別のワーク変数

mt[]更新

▼ mt更新



更新式 ： mt[kk+624] = f(mt[kk], mt[kk+1])

mt[kk+1]を軸に見ると計算量が減る
（mt[kk]は1bitしか関わらないので）

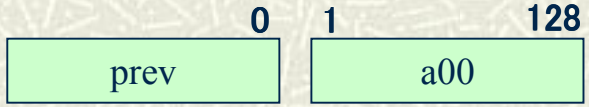
mt[kk+624] = f(mt[kk+1]) とみなす。 ※ mt[kk+M] は別途

(y>>1)^mag01[y&0x1UL]までの計算

▼ y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);

```
a00 = spu_sel( a00, prev, 0x1 );
a00 = spu_rlqwbyte( a00, 15 );
a00 = spu_rlqw( a00, 7 );
```

最上位ビット:



▼ (y >> 1) ^ mag01[y & 0x1UL];

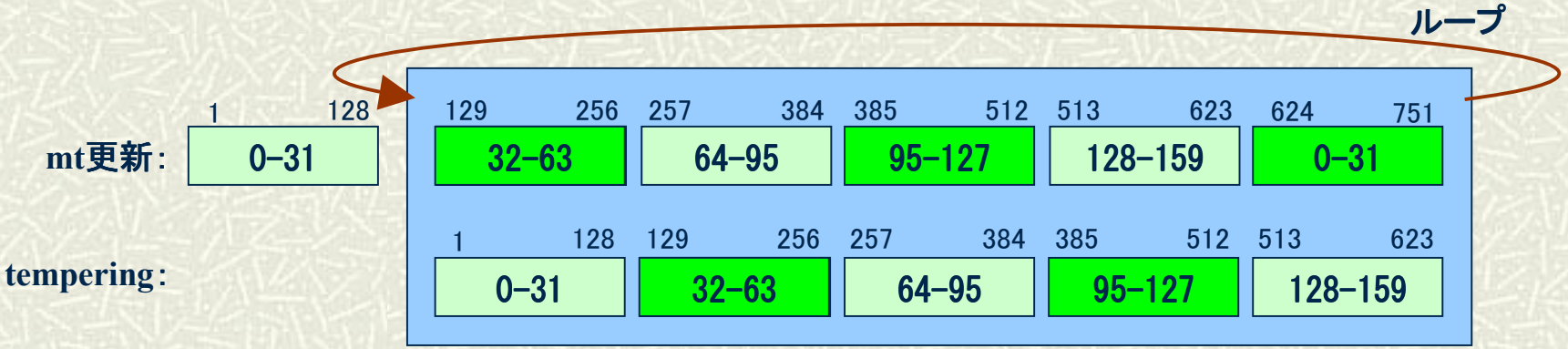
```
a02 = spu_xor( a02, a31 );
a03 = spu_xor( a03, a31 );
a06 = spu_xor( a06, a31 );
a11 = spu_xor( a11, a31 );
a15 = spu_xor( a15, a31 );
a17 = spu_xor( a17, a31 );
a18 = spu_xor( a18, a31 );
a23 = spu_xor( a23, a31 );
a24 = spu_xor( a24, a31 );
a26 = spu_xor( a26, a31 );
a27 = spu_xor( a27, a31 );
a28 = spu_xor( a28, a31 );
a29 = spu_xor( a29, a31 );
a30 = spu_xor( a30, a31 );
```

mag01[2]={0x0UL, 0x9908b0df};

「a31,a00,・・・,a30」の組に
結果が入る

mt[]更新とtempering計算

▼ mt更新とtempering計算のパイプライン



:レジスタ・セットA　32レジスタ

:レジスタ・セットB　32レジスタ

mt更新　　：odd命令が多い

tempering：even命令が多い

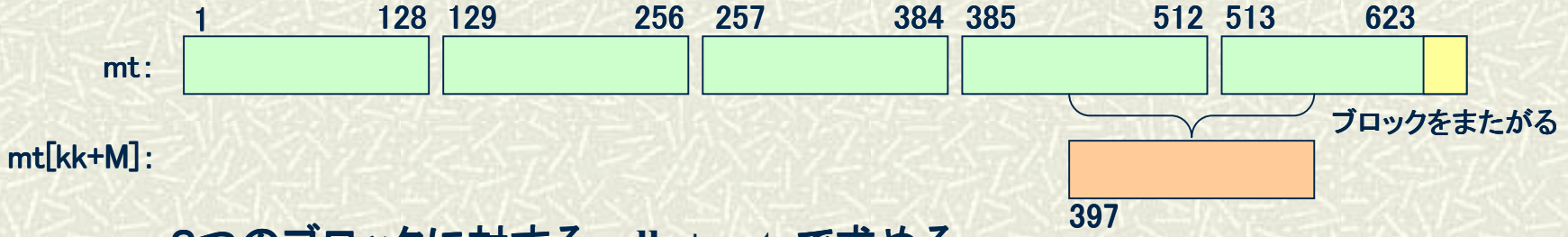
互いに依存関係がなければ
ペアリングしやすい

◎ ループの終端で“0-31” のレジスタセットが異なる

→ ループ内を10セット分（上図の倍）にアンロール

（※ 時間切れで実装できず。セットBからセットAへの代入コストを払っている。）

mt[kk+M]の算出

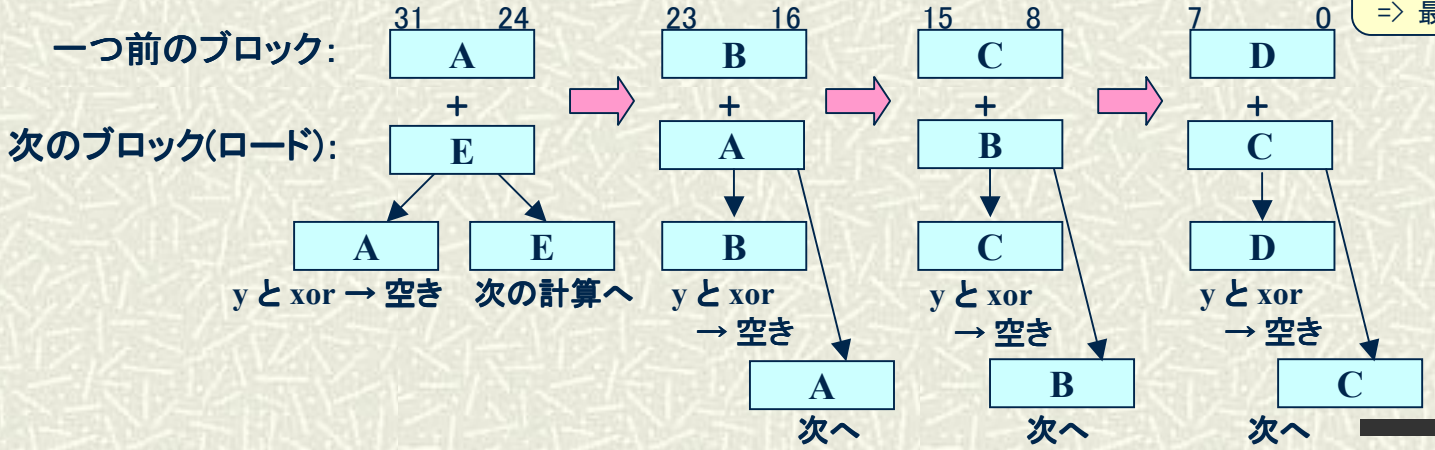


・2つのブロックに対する selb + rot で求める。

・理想的には，32個のレジスタを2セット使ってメモリロードを減らしたい。
→ レジスタが足りない

・レジスタ8個（8ビット分）×５セット を うまく回すことで対処

mt[513]～mt[623] では
特異処理となる。
=> 最適化が不十分で遅い・・・



tempering

▼ tempering計算

y ^= (y >> 11)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
											0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

y ^= (y << 7) & 0x9d2c5680UL

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
7			10	11	12		14			17		19	20				24		26		28	29		31							
1	0	0	1	1	1	0	1	0	0	1	0	1	1	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0

y ^= (y << 15) & 0xefc60000UL

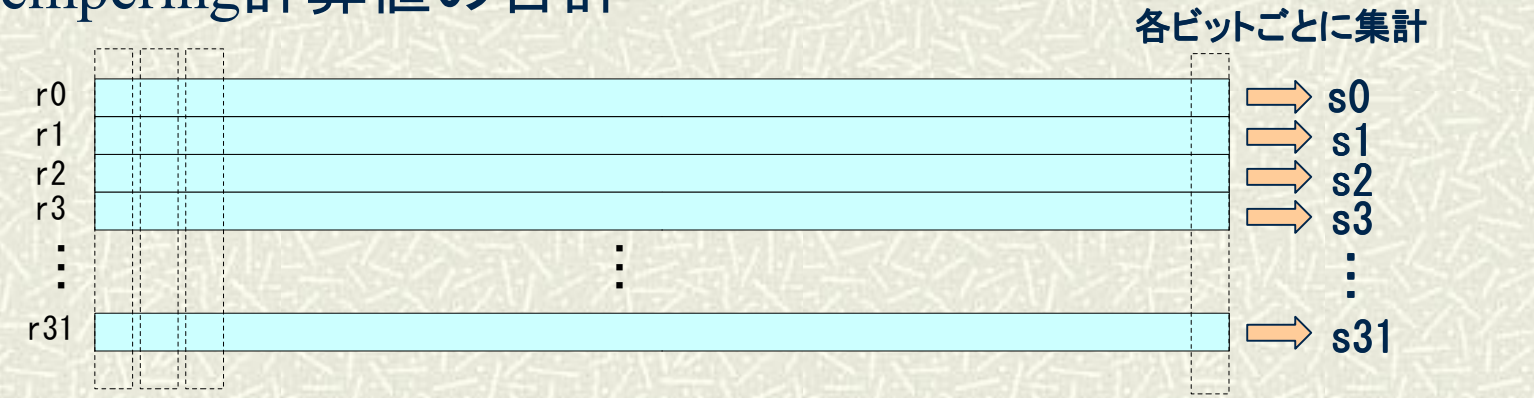
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
15	16	17		19	20	21	22	23	24				28	29																	
1	1	1	0	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

y ^= (y >> 18)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
																		0	1	2	3	4	5	6	7	8	9	10	11	12	13

合計値

▼ tempering計算値の合計



さらに，ブロックごとにs0～s31を集計

```
total0 += s0
total1 += s1
total2 += s2
total3 += s3
   ⋮
total31 += s31
```

total0 ～ totoal15（上位ビット）は short で集計

total16 ～ totoal31（下位ビット）は int で集計

→ ６レジスタ使用

最終結果は，最後に各ビットをシフトしながら合計する

SUMコスト： cntb×32, sumb×20, shuffle×12, a×6 (even=58, odd=12)

アセンブラ関数

128個のレジスタを全て使うためにオールアセンブラの関数を記述

▼ エントリポイント

ラベル + extern で記述

```
extern RESULT_SUM sum_rand_asm( UINTV loop_cnt, UINTV preve );
__asm__ (
            "sum_rand_asm:¥n¥t"
```

▼ prologue

レジスタ $80～$127 を退避

```
"stqd       $80,-16($sp)¥n¥t"
"stqd       $81,-32($sp)¥n¥t"
"stqd       $82,-48($sp)¥n¥t“
・・・
"stqd       $127,-768($sp)¥n¥t"
```

アセンブラ関数

▼ 引数

$3 から(引数の数だけ)順番に格納される。

```
$0 = $lr 戻りアドレス
$1 = $sp スタックポインタ
$2       環境ポインタ（フレームポインタ？）
```

▼ 戻り値

$3 に入れる。構造体の場合は，メンバの数だけ$3から順番に格納する。

▼ epilogue

退避したレジスタ $80～$127 を戻して，戻りアドレスへジャンプ

```
"hbr         FUNCTION_END,$lr¥n¥t"
"lqd         $80,-16($sp)¥n¥t"
"lqd         $81,-32($sp)¥n¥t“
 . . .
"lqd         $127,-768($sp)¥n¥t"
"FUNCTION_END:¥n¥t"
"bi          $lr¥n¥t"
```

End