完全ハッシュ関数の
ハードウェア向け実装について

九州大学
松永　裕介

（最小）完全ハッシュ関数
（Minimal）Perfect Hash Function

- 入力： 集合Uとその部分集合$S \subseteq U$
- 完全ハッシュ関数(perfect hash function: PHF) $h: U \to [0, k\ -1]$とは

$$\forall s_i, s_j \in S\ (i \neq j)\ h(s_i) \neq h(s_j)$$

を満たす関数のことである.

- $k = |S|$の時、最小完全ハッシュ関数と呼ばれる。
- $u \notin S$に対しては$h(u)$の値はなんでもよい。

インデックス生成関数 (IGF)

- $F\colon B^n \to \{0, 1, 2, \cdots, k\}$
  - 2値のn入力、(k + 1)値出力の関数
- 登録ベクタ（集合）: $v_1, v_2, \cdots, v_k \in B^n$
- $$F(v) = \begin{cases} i & : v = v_i \\ 0 & : \text{otherwise} \end{cases}$$

SASAO, T. “Index Generation Functions: Recent Developments.” In *the 41st IEEE International Symposium on Multiple-Valued Logic* (2011)

インデックス生成関数の例

x_1	x_2	x_3	x_4	インデックス
0	1	1	0	1
0	0	1	0	2
1	1	0	1	3
0	1	1	1	4

0010 ⇒ 2

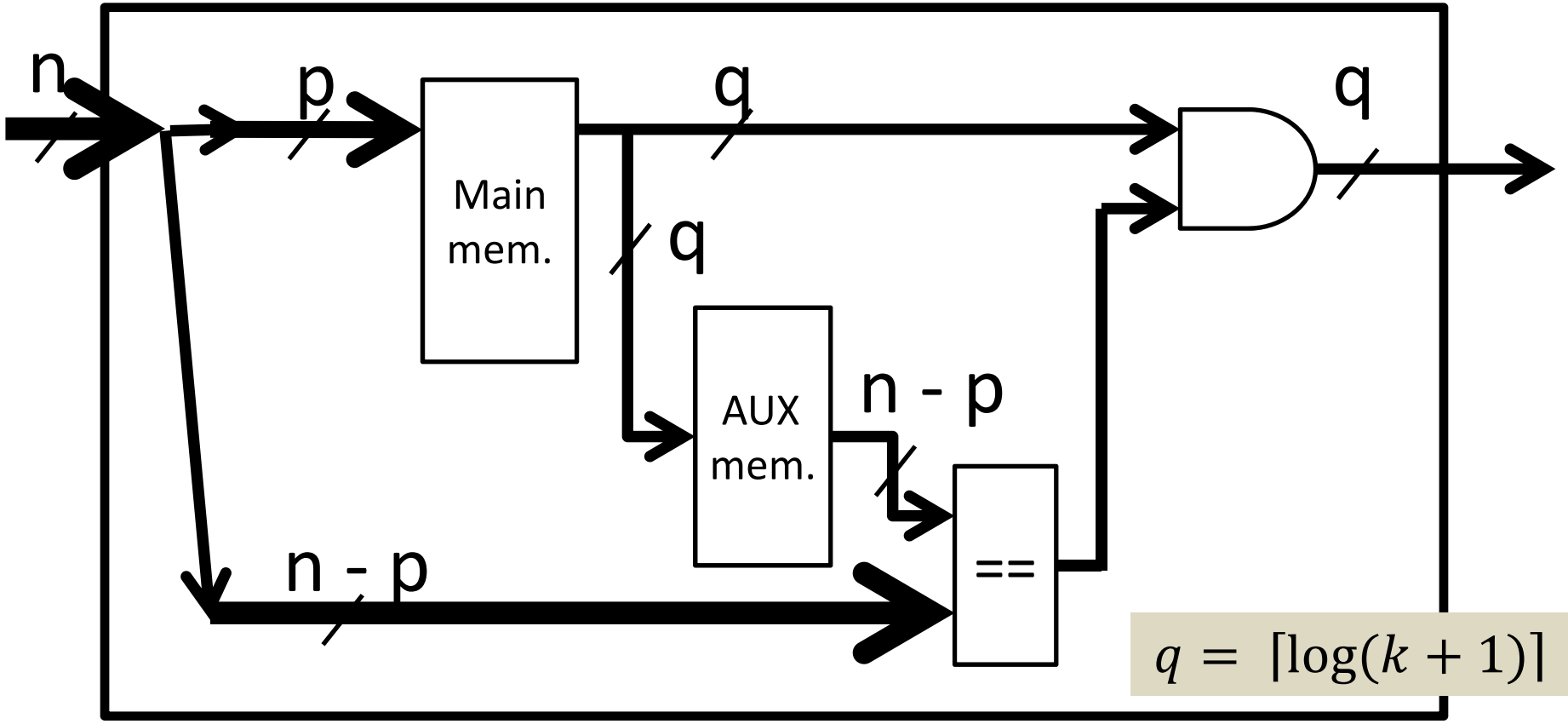
1010 ⇒ 0 (登録されていない)

仮定:

- ビットベクタとインデックスの間に規則性はない。
- データの数(キーの数)は入力空間に比べればはるかに少ない。

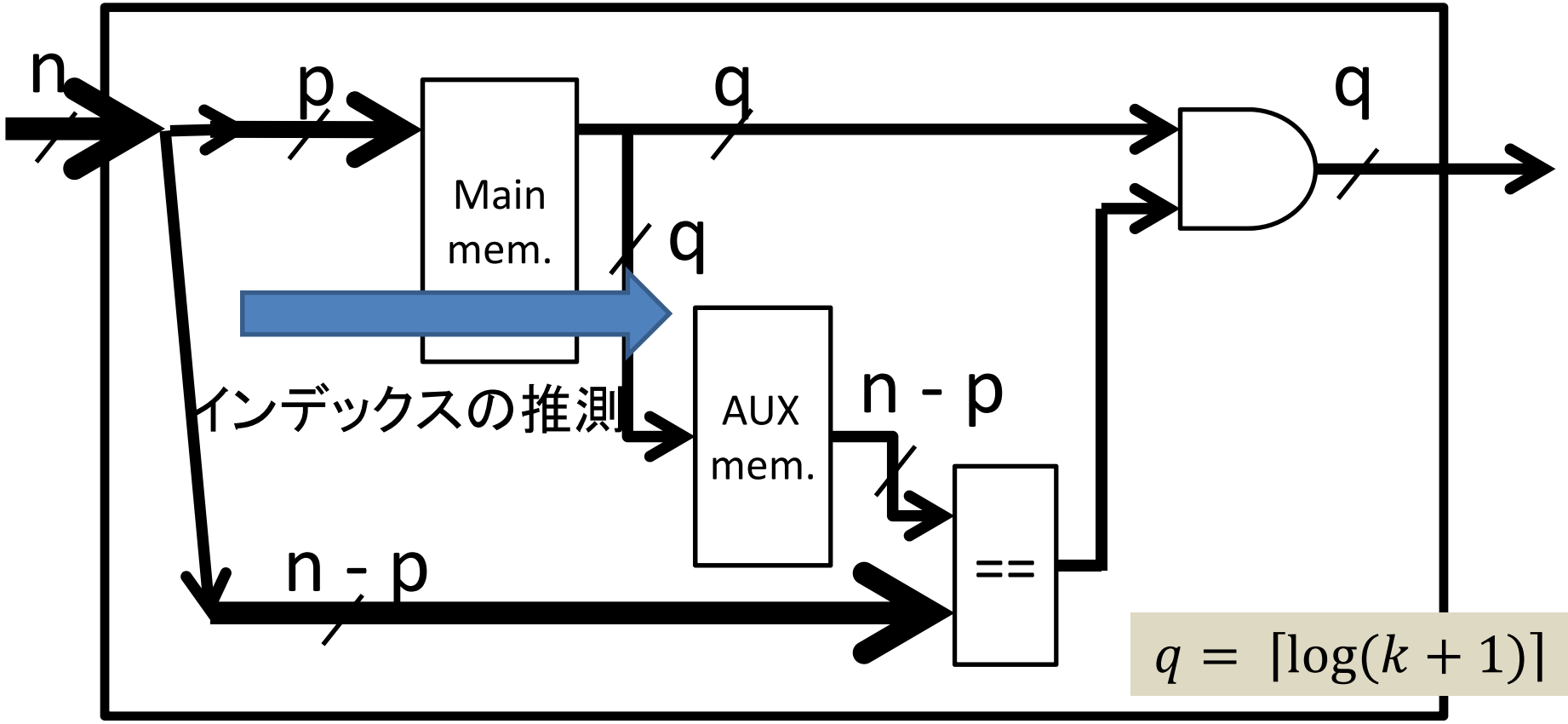
笹尾のインデックス生成器(IGU)

インデックス生成関数の一つの構成法



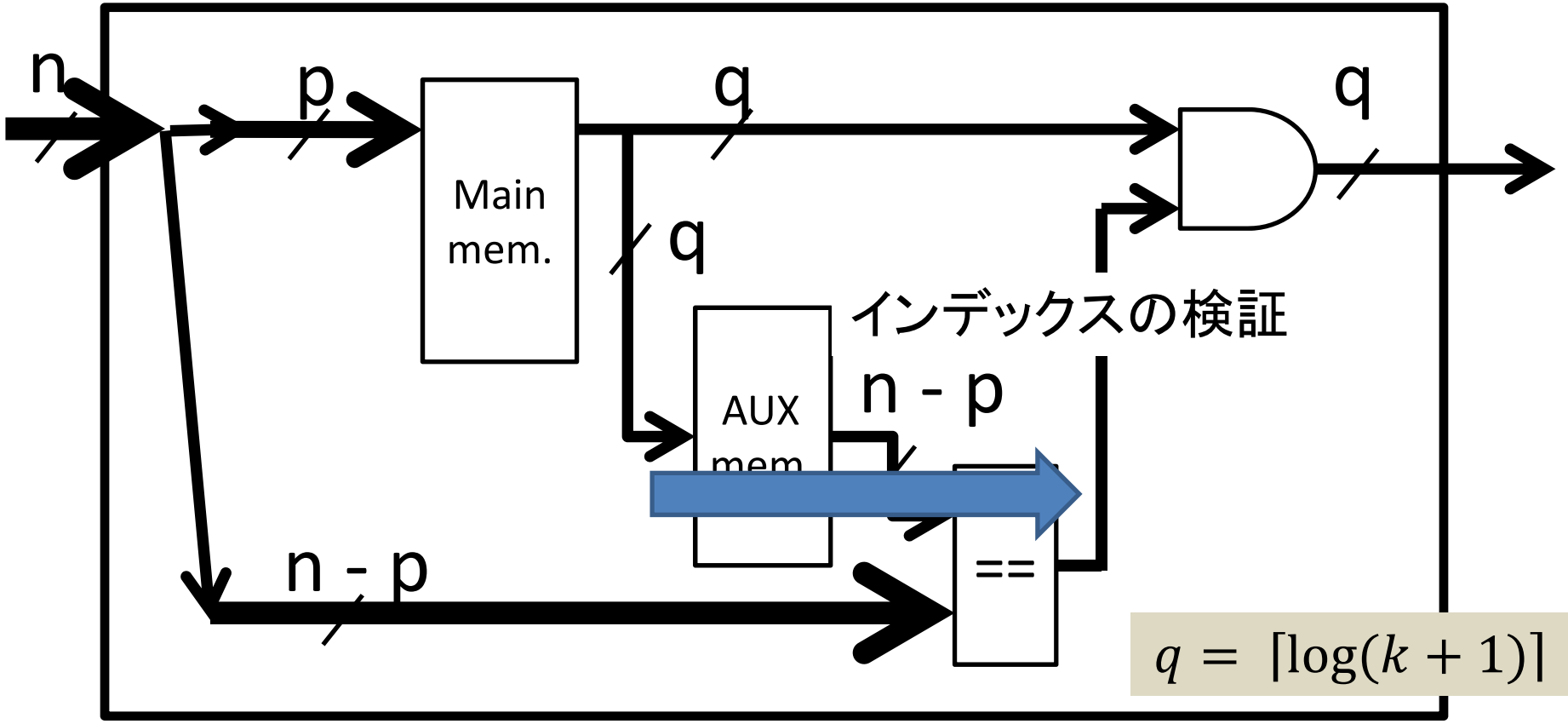
笹尾のインデックス生成器(IGU)

インデックス生成関数の一つの構成法



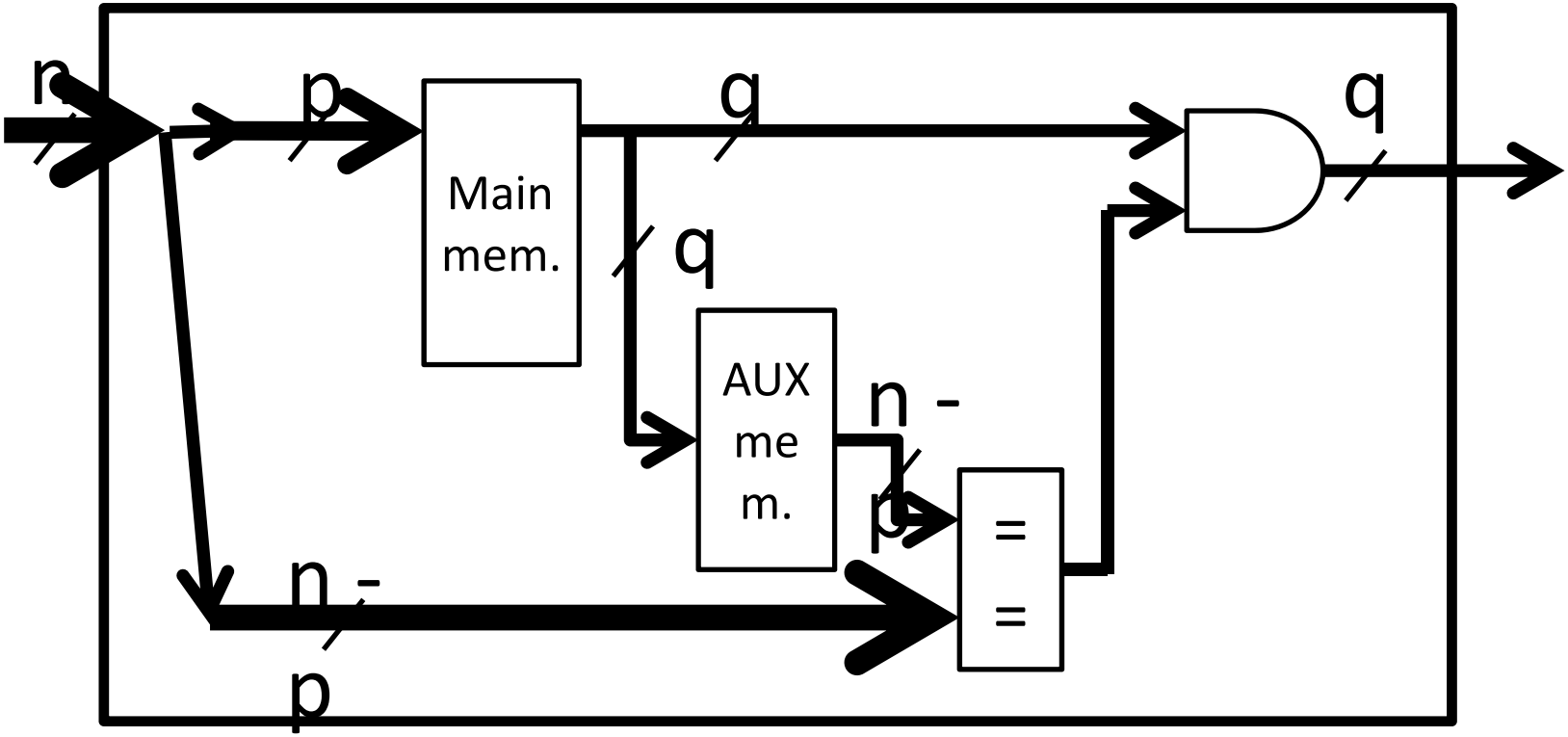
笹尾のインデックス生成器(IGU)

インデックス生成関数の一つの構成法



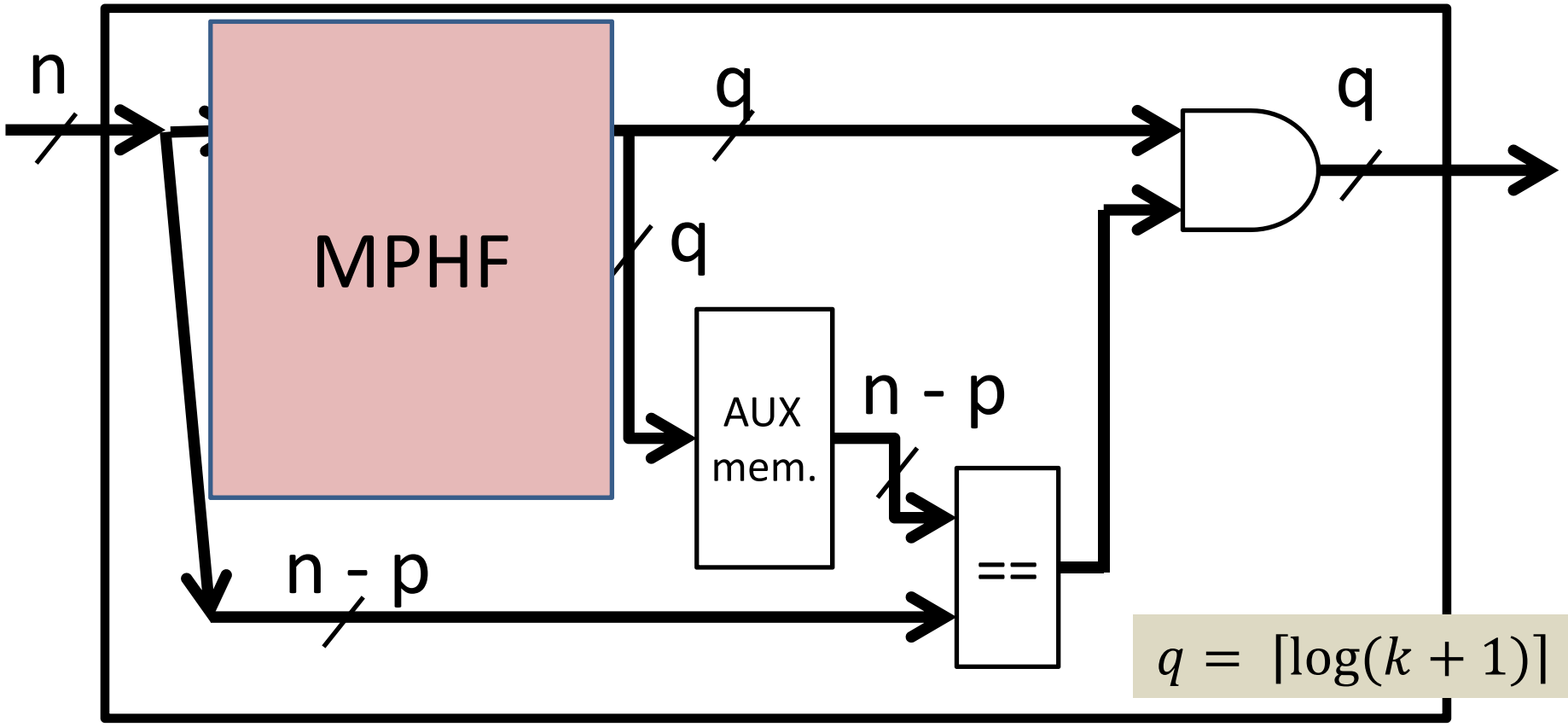
笹尾のインデックス生成器(IGU)

- Main メモリの出力は衝突してはならない
- pの値は qよりもかなり大きくなる。



MPHF と IGU

最小完全ハッシュ関数（MPFH）
　= 不完全定義インデックス生成関数



インデックス生成器/完全ハッシュ関数のハードウェア化の意義

- 従来の論理合成は論理式ベース
  - よい論理式からよい論理回路を作る。
- では、入出力の関係のみ与えられた関数をどうやって設計する？
- 基本は表引きメモリ

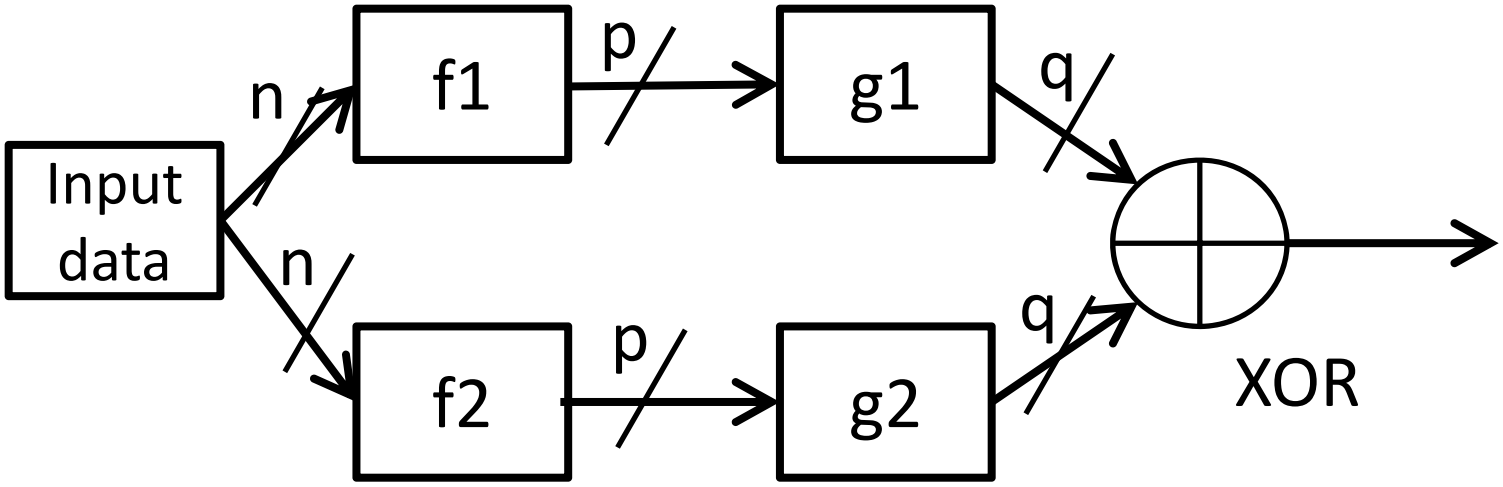
- 問題はメモリのサイズ
  - 完全ハッシュ関数用のソフトウェアアルゴリズムをハードウェア用にしたものがよいか、専用の構成法を考えたほうがよいか？

完全ハッシュ関数の構成法（1）

- Z. J. Czech, G. Havas, and B. S. Majewski. “An optimal algorithm for generating minimal perfect hash functions”, Information Processing Letters,1992
- Botelho, F.C., Pagh, R. and Ziviani, N. "Simple and Space-Efficient Minimal Perfect Hash Functions“, WADS07, 2007

最小完全ハッシュ関数の構成法(2)

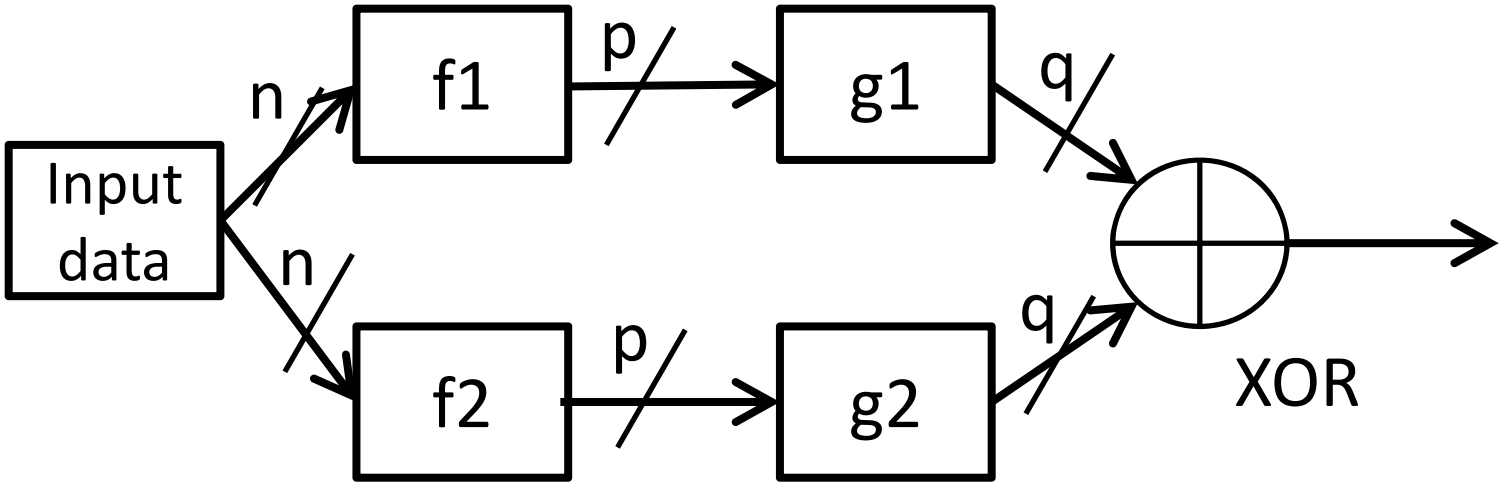
- f1, f2: 入力ハッシュ関数 (シグネチャ関数)
  - 論理回路で実現される
- g1, g2: マッピング関数
  - メモリを用いたルックアップテーブルで実現される



最小完全ハッシュ関数の構成法(2)

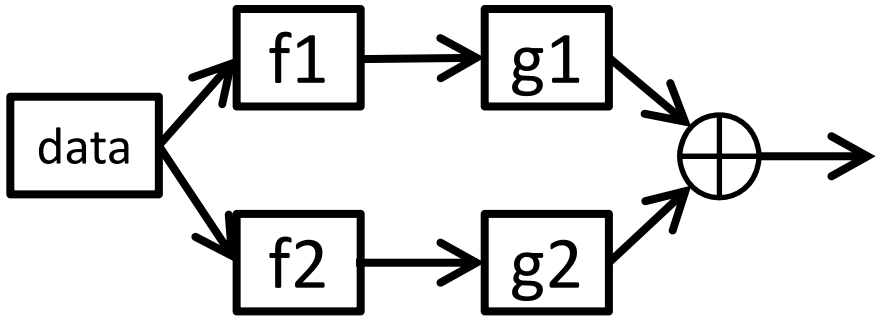
- f1, f2: 入力ハッシュ関数 (シグネチャ関数)
  - 論理回路で実現される
- g1, g2: マッピング関数
  - メモリを用いたルックアップテーブルで実現される

例えば n = 20 ～ 30
p = 10



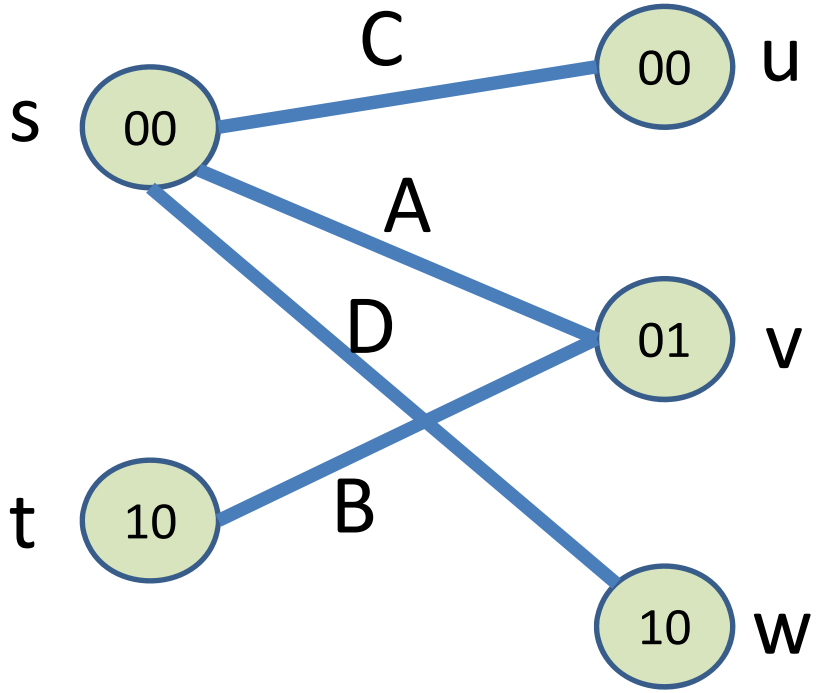
Example

Data	Index	F1	F2
A	0 (00)	00	01
B	1 (01)	10	01
C	2 (10)	00	00
D	3 (11)	00	10



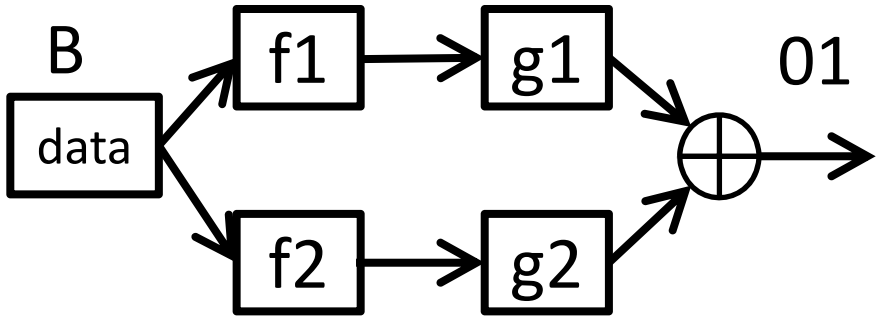
	g1
00 (s)	
01	
10 (t)	
11	

	g2
00 (u)	
01 (v)	
10 (w)	
11	



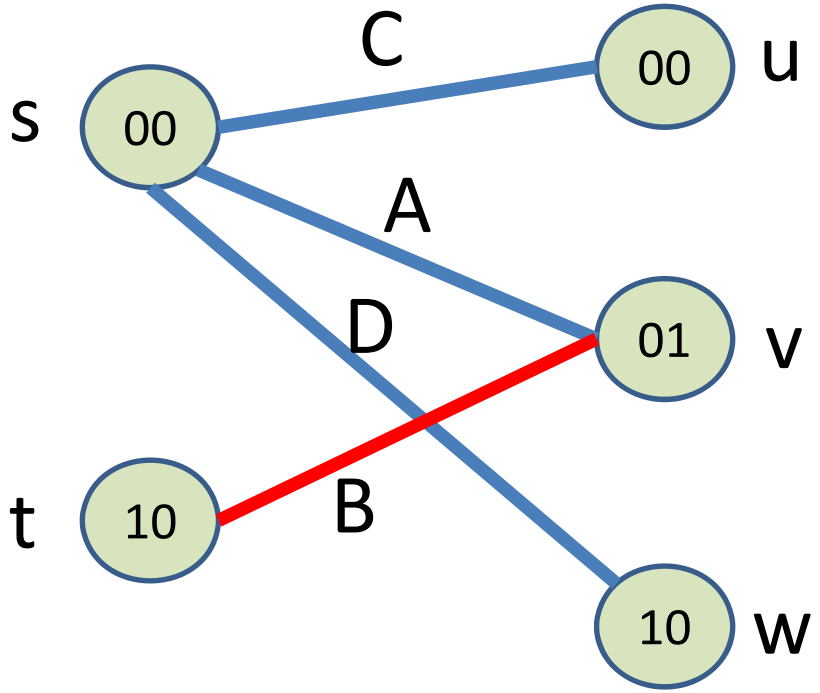
Example

Data	Index	F1	F2
A	0 (00)	00	01
B	1 (01)	10	01
C	2 (10)	00	00
D	3 (11)	00	10



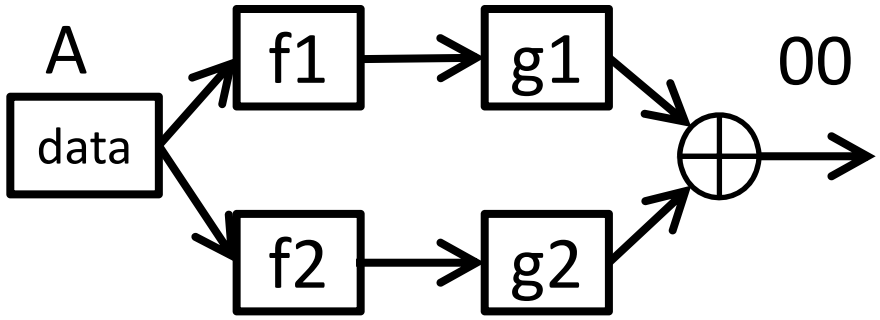
	g1
00 (s)	
01	
10 (t)	00
11	

	g2
00 (u)	
01 (v)	01
10 (w)	
11	



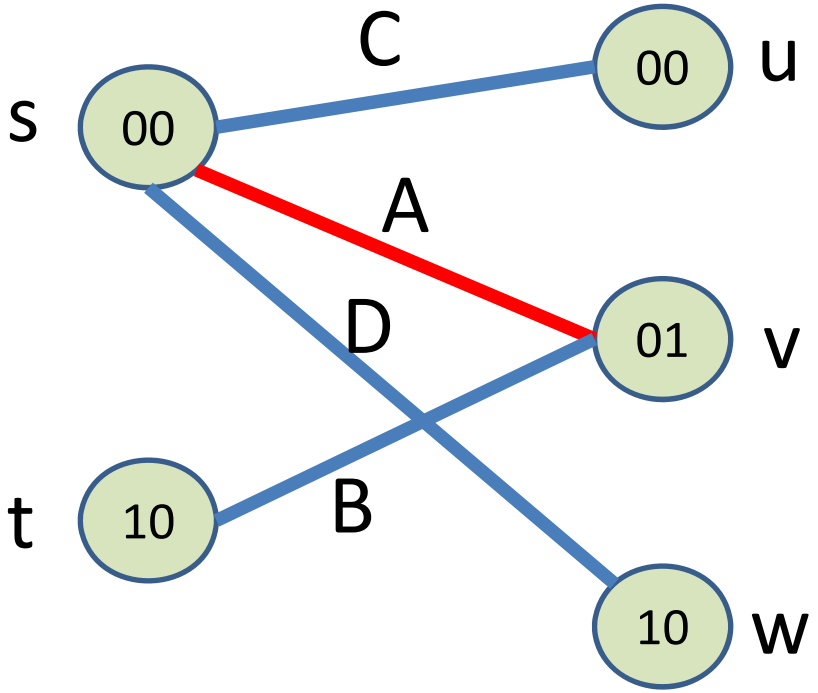
Example

Data	Index	F1	F2
A	0 (00)	00	01
B	1 (01)	10	01
C	2 (10)	00	00
D	3 (11)	00	10



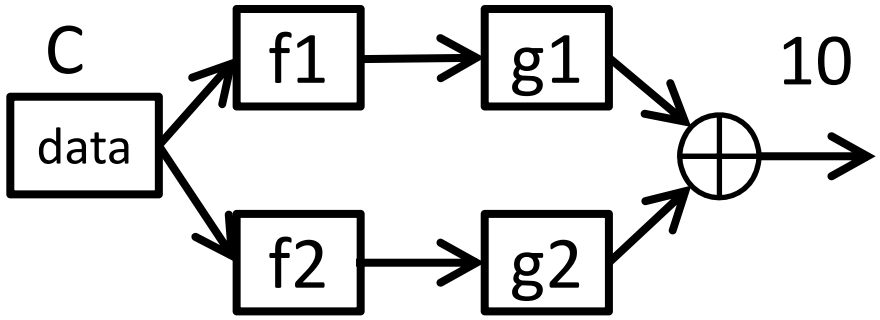
	g1
00 (s)	01
01	
10 (t)	00
11	

	g2
00 (u)	
01 (v)	01
10 (w)	
11	



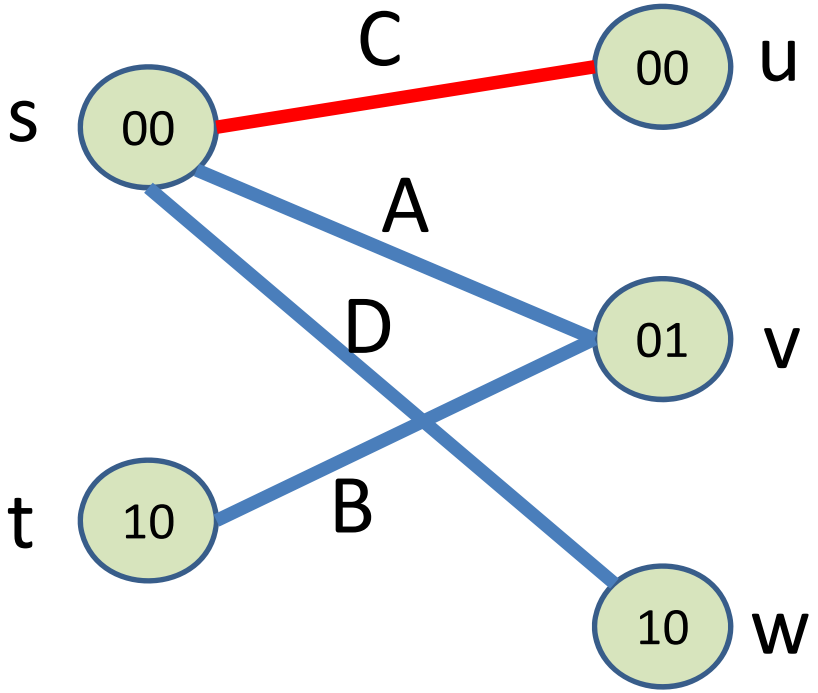
Example

Data	Index	F1	F2
A	0 (00)	00	01
B	1 (01)	10	01
C	2 (10)	00	00
D	3 (11)	00	10



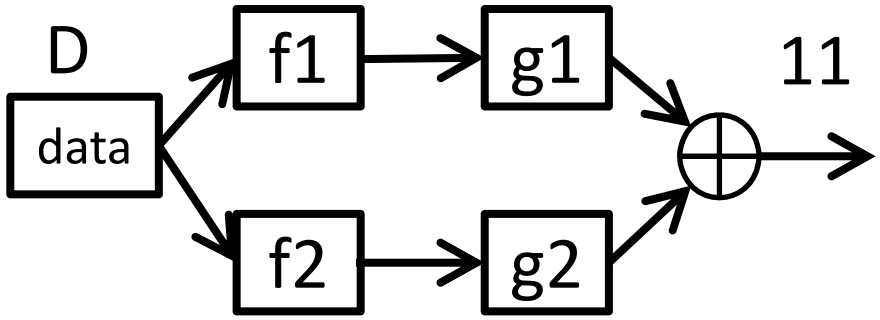
	g1
00 (s)	01
01	
10 (t)	00
11	

	g2
00 (u)	11
01 (v)	01
10 (w)	
11	



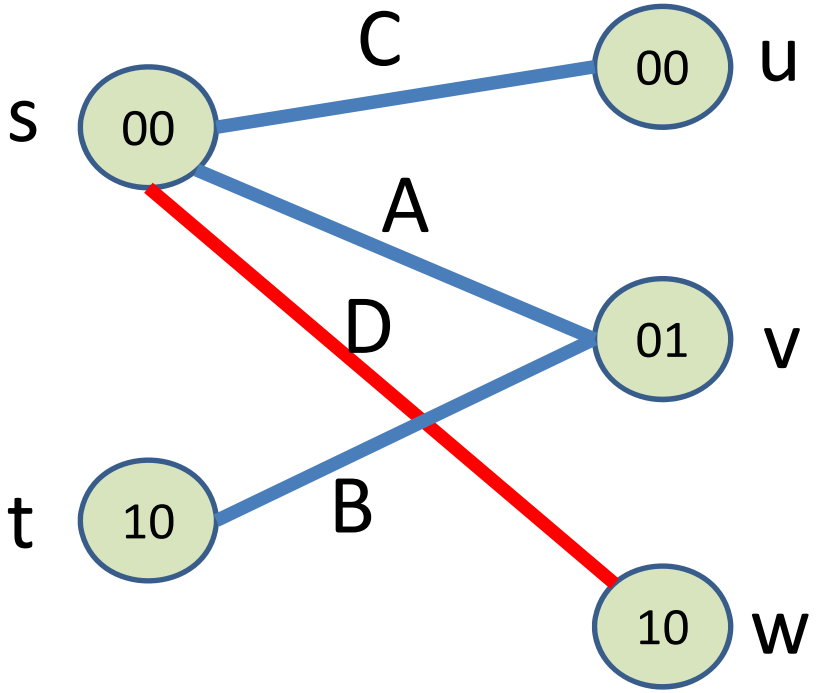
Example

Data	Index	F1	F2
A	0 (00)	00	01
B	1 (01)	10	01
C	2 (10)	00	00
D	3 (11)	00	10



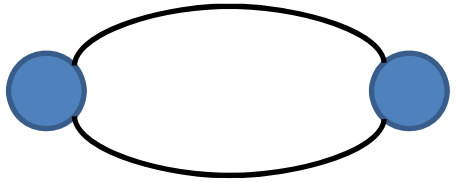
	g1
00 (s)	01
01	-
10 (t)	00
11	-

	g2
00 (u)	11
01 (v)	01
10 (w)	10
11	-

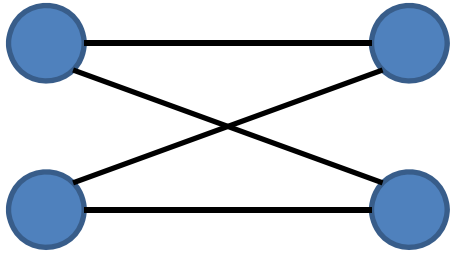


入力ハッシュ関数の制約

- 生成されるグラフが「単純」でなければならない
  - ハッシュの衝突を防ぐため

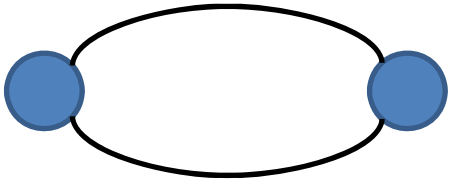


- 生成されるグラフが「非巡回」でなければならない
  - g1とg2に矛盾しない割り当てを保証するため



入力ハッシュ関数の制約

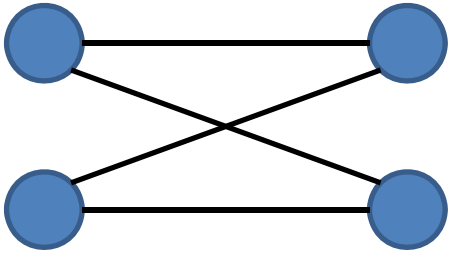
- 生成されるグラフが「単純」でなければならない
  - ハッシュの衝突を防ぐため



- 生成されるグラフが「非巡回」でなければならない
  - g1とg2に矛盾しない割り当てを保証するため

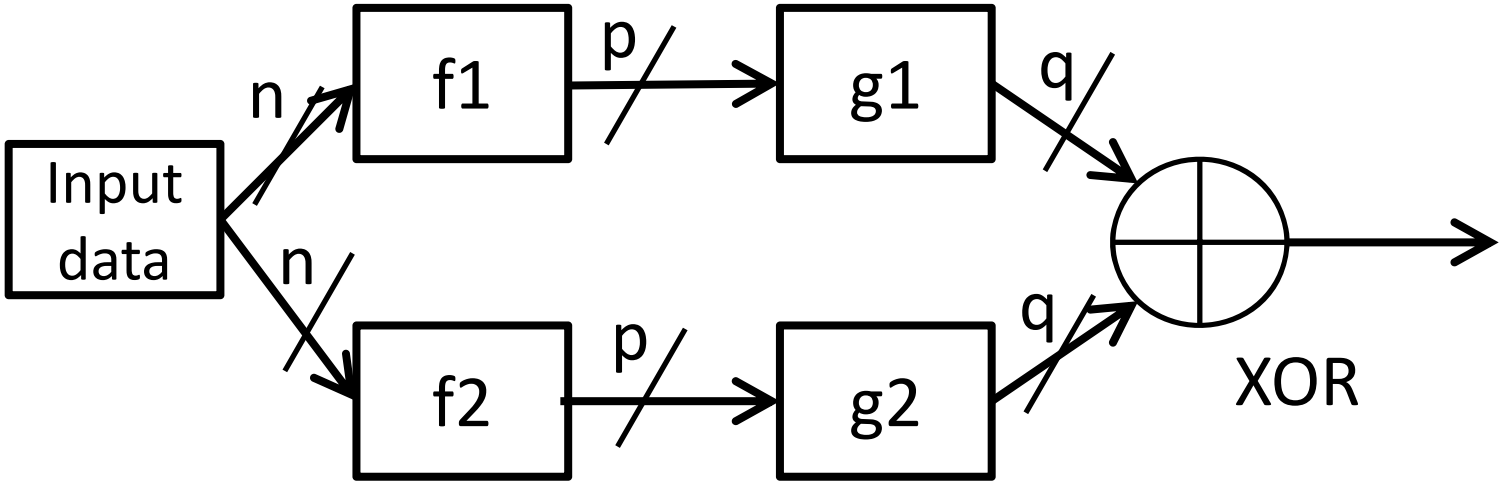
問題：

そのような制約を満たす入力ハッシュ関数をどうやって見つける?



目的関数

- nとqの値はあらかじめ決まっている。
- 出力マッピング関数(g1, g2)のメモリ量はpの値の指数乗に比例する。
- ⇒pの値を最小化したい。



ランダムサンプリングによる入力ハッシュ関数の評価

- ランダムに生成した入力ハッシュ関数(の組)が最小完全ハッシュ関数の構成条件を満たす確率を評価

	r = 2			r = 3		
2^p	成功	条件1	条件2	成功	条件1	条件2
512	0	8666	10000	9955	45	45
1024	2539	3893	7461	9994	6	6
2048	8669	1154	1331	9999	1	1

キーの個数: 1024

ランダムサンプリングによる入力ハッシュ関数の評価

R = 2 の場合は入力ハッシュ関数の良し悪しにセンシティブ

R = 3 の場合はキーの個数の半分の大きさのテーブル×3で実現可能

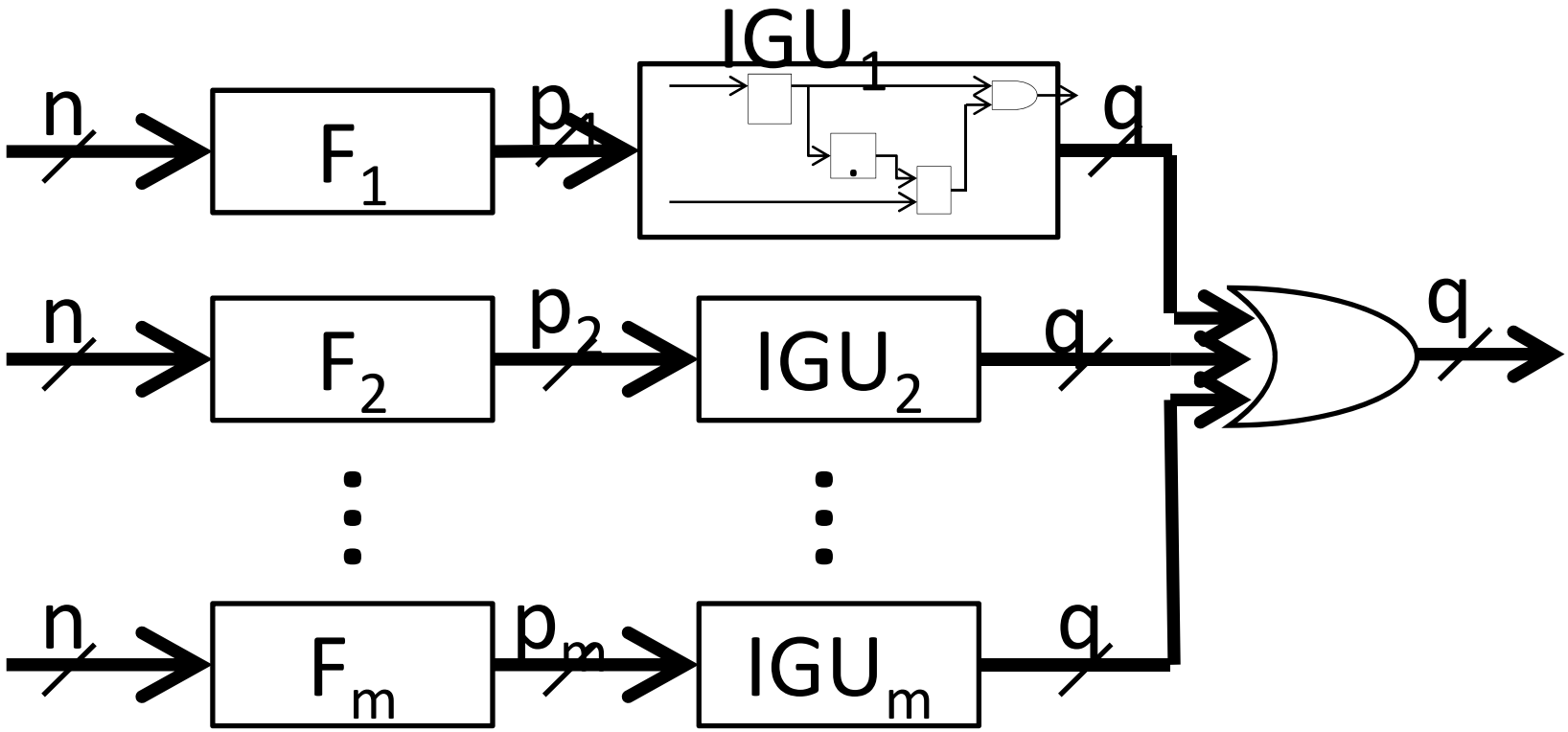
す確率を評価

	r = 2			r = 3		
2^p	成功	条件1	条件2	成功	条件1	条件2
512	0	8666	10000	9955	45	45
1024	2539	3893	7461	9994	6	6
2048	8669	1154	1331	9999	1	1

キーの個数: 1024

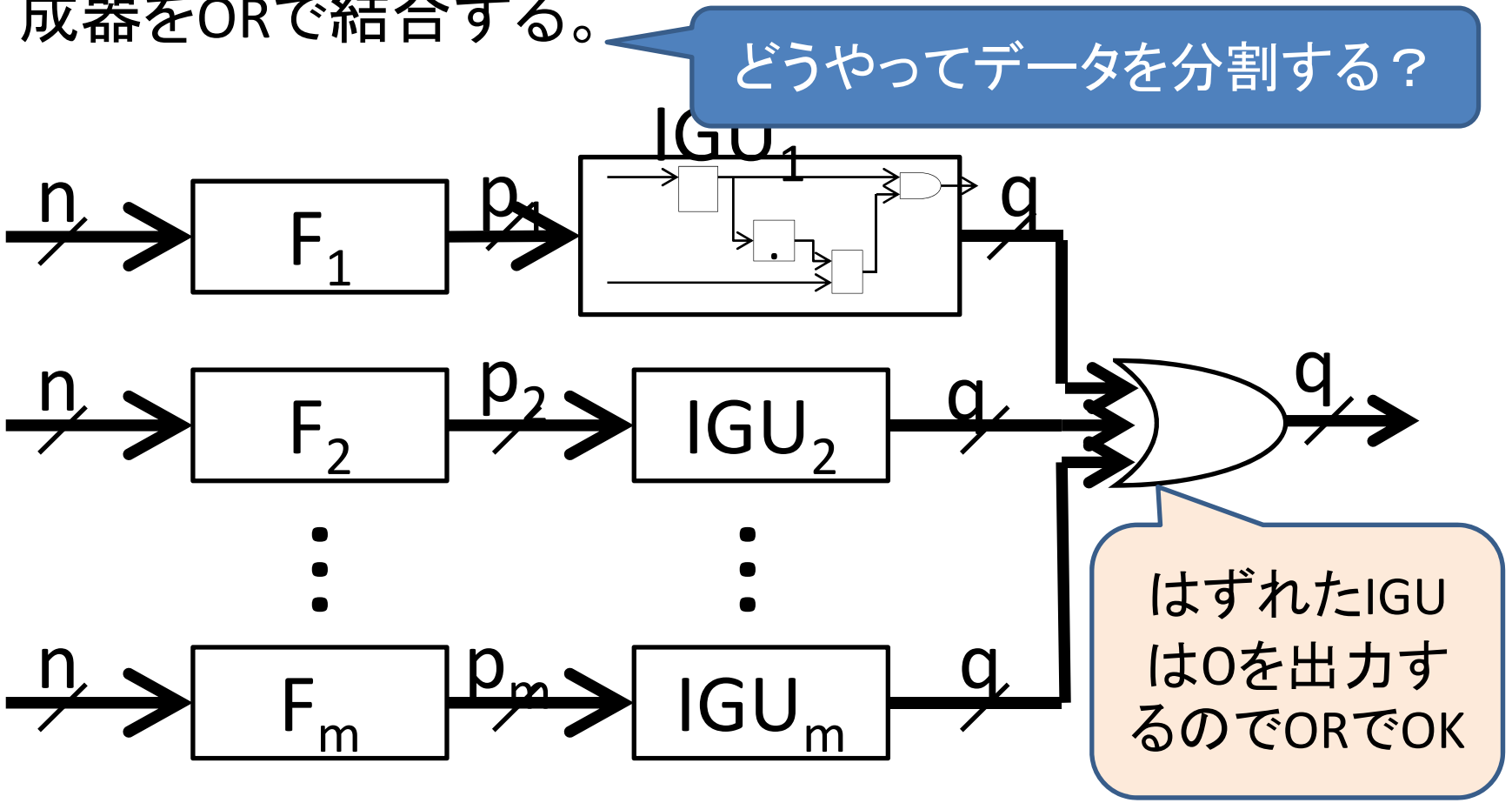
並列インデックス生成器

- 異なる入力ハッシュ関数を持った複数のインデックス生成器をORで結合する。



並列インデックス生成器

- 異なる入力ハッシュ関数を持った複数のインデックス生成器をORで結合する。

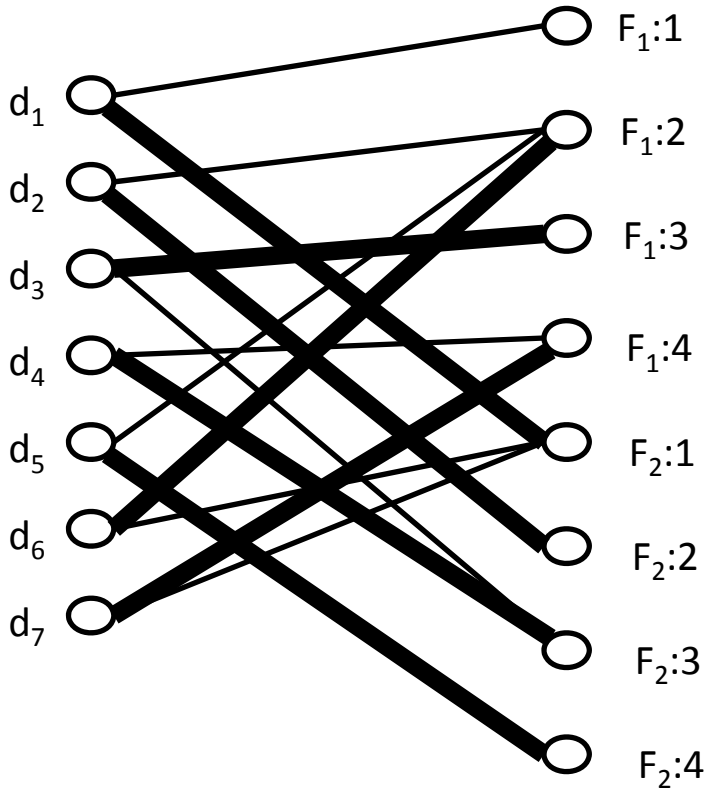


コンフリクトフリー分割

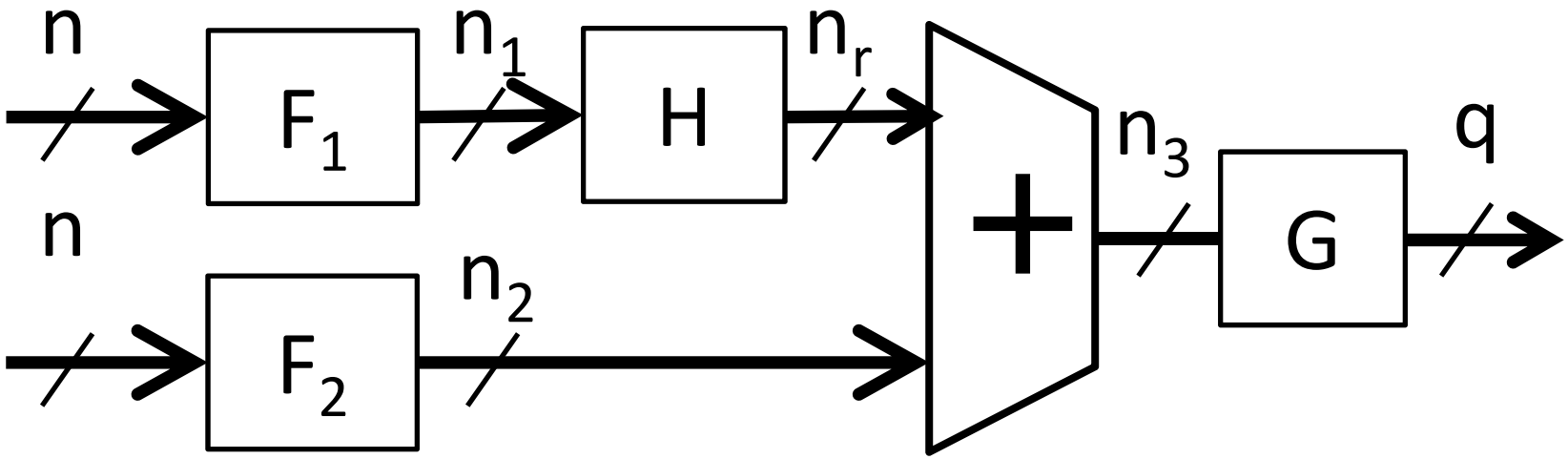
	F_1	F_2
d_1	1	1
d_2	2	2
d_3	3	3
d_4	4	3
d_5	2	4
d_6	2	1
d_7	4	1

コンフリクトフリー分割

	F_1	F_2
d_1	1	1
d_2	2	2
d_3	3	3
d_4	4	3
d_5	2	4
d_6	2	1
d_7	4	1



行シフト分解



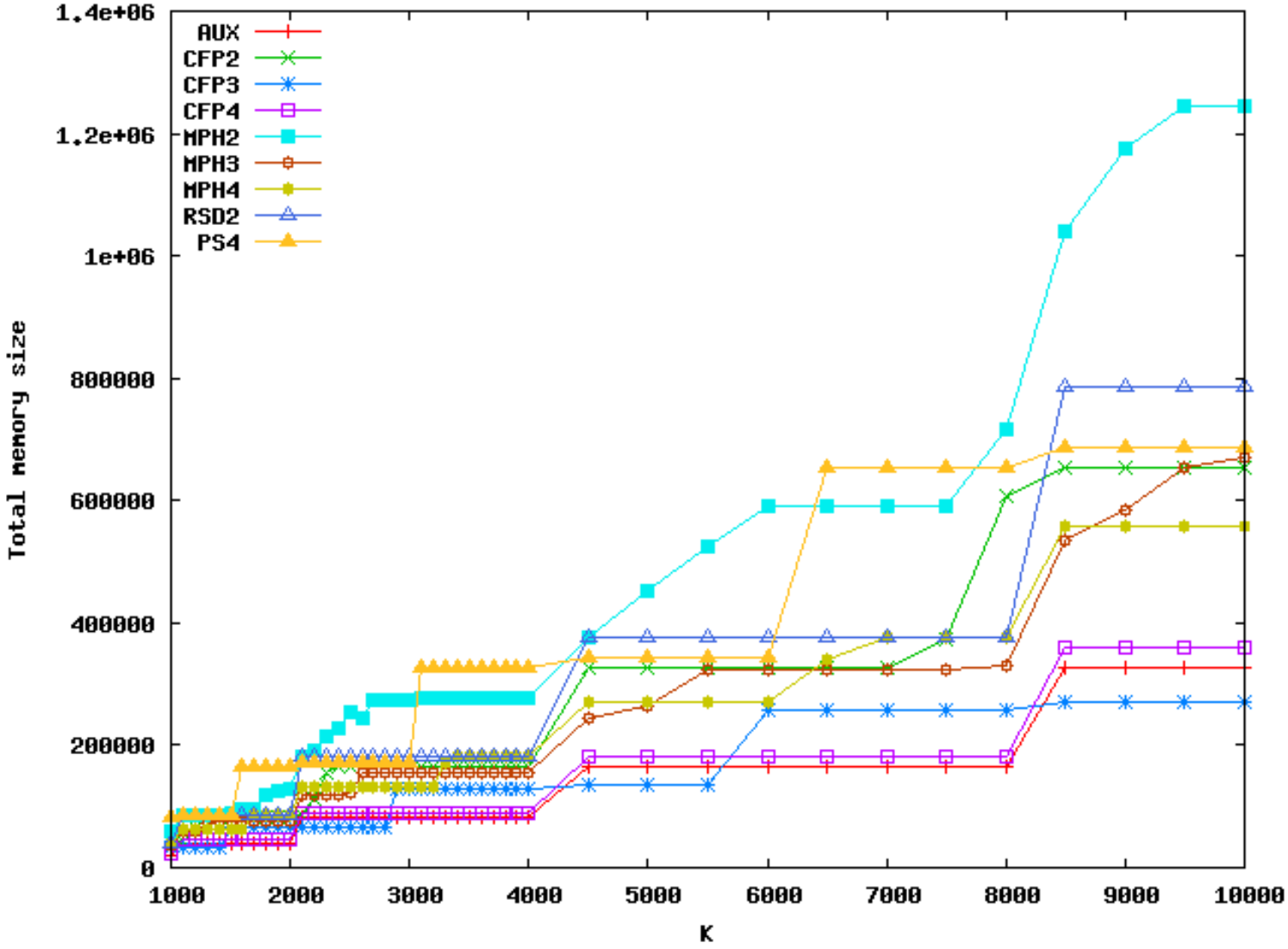
- T. Sasao. “Row-shift decompositions for index generation functions”, In *Design, Automation Test in Europe Conference Exhibition(DATE), 2012*, pp. 1585–1590, 2012

評価実験

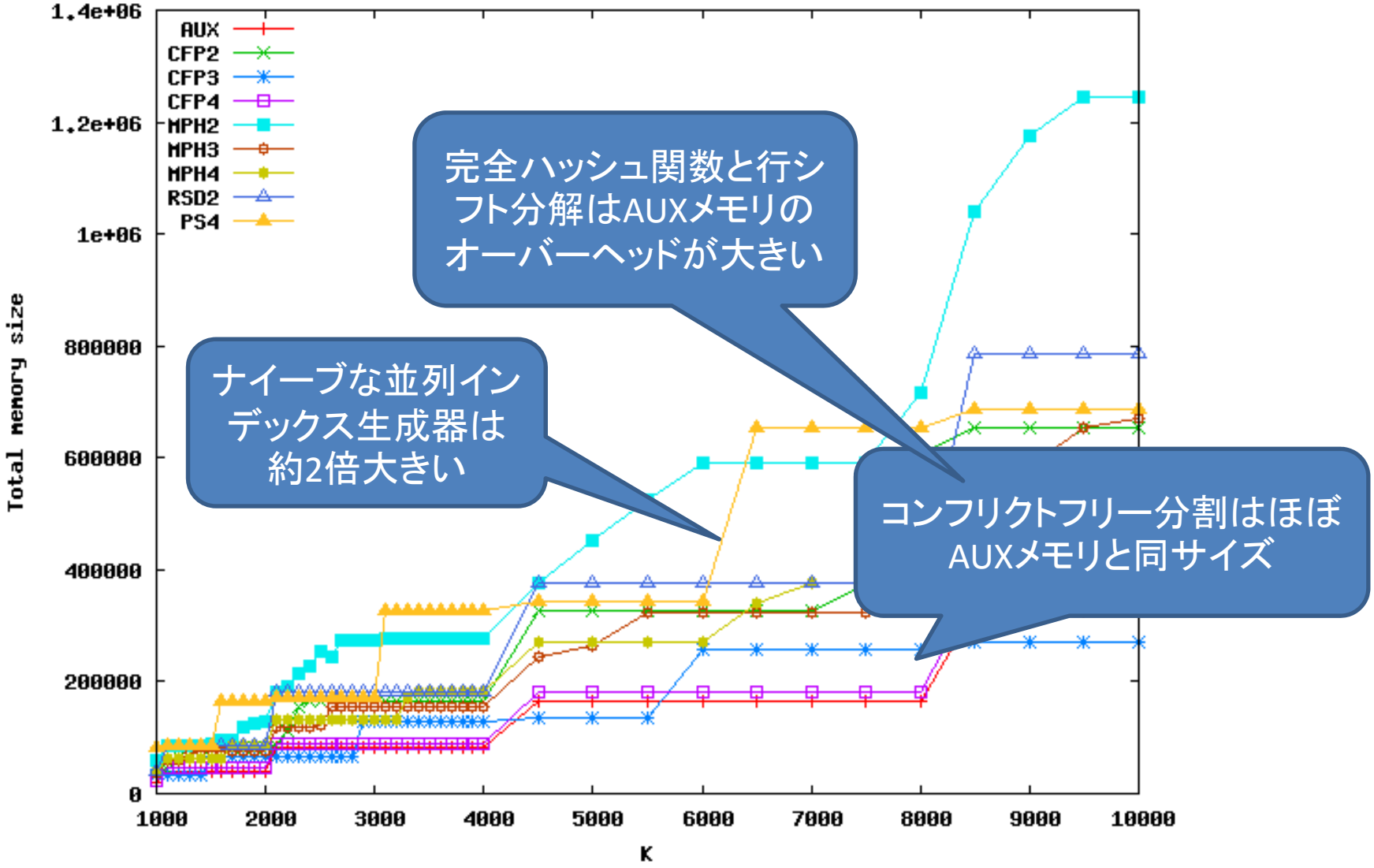
- コンフリクトフリー分割を用いて並列インデックス生成器
- 完全ハッシュ関数(ランダムグラフ)
- 行シフト分解
- ナイーブな並列インデックス生成器
- AUXメモリ単体($=2^q \times n$)

- のメモリサイズを比較
  - 入力のビット幅は20ビット
  - キーの要素数は 1,000 から　10,000

実験結果

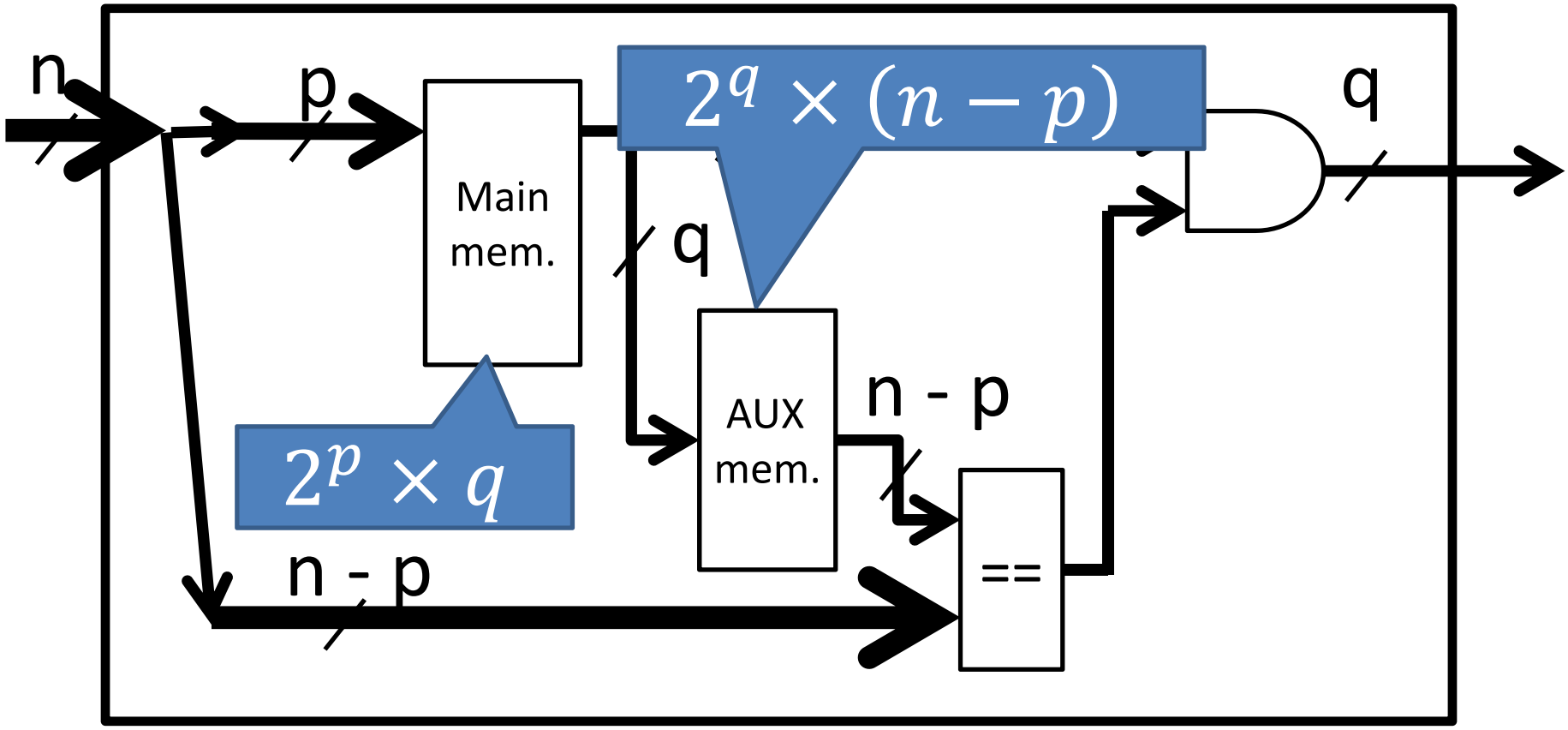

AUX
CFP2
CFP3
CFP4
MPH2
MPH3
MPH4
RSD2
PS4
Total memory size
K
1.4e+06
1.2e+06
1e+06
800000
600000
400000
200000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

実験結果


AUX
CFP2
CFP3
CFP4
MPH2
MPH3
MPH4
RSD2
PS4
Total memory size
K
1.4e+06
1.2e+06
1e+06
800000
600000
400000
200000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
完全ハッシュ関数と行シフト分解はAUXメモリのオーバーヘッドが大きい
ナイーブな並列インデックス生成器は約2倍大きい
コンフリクトフリー分割はほぼAUXメモリと同サイズ

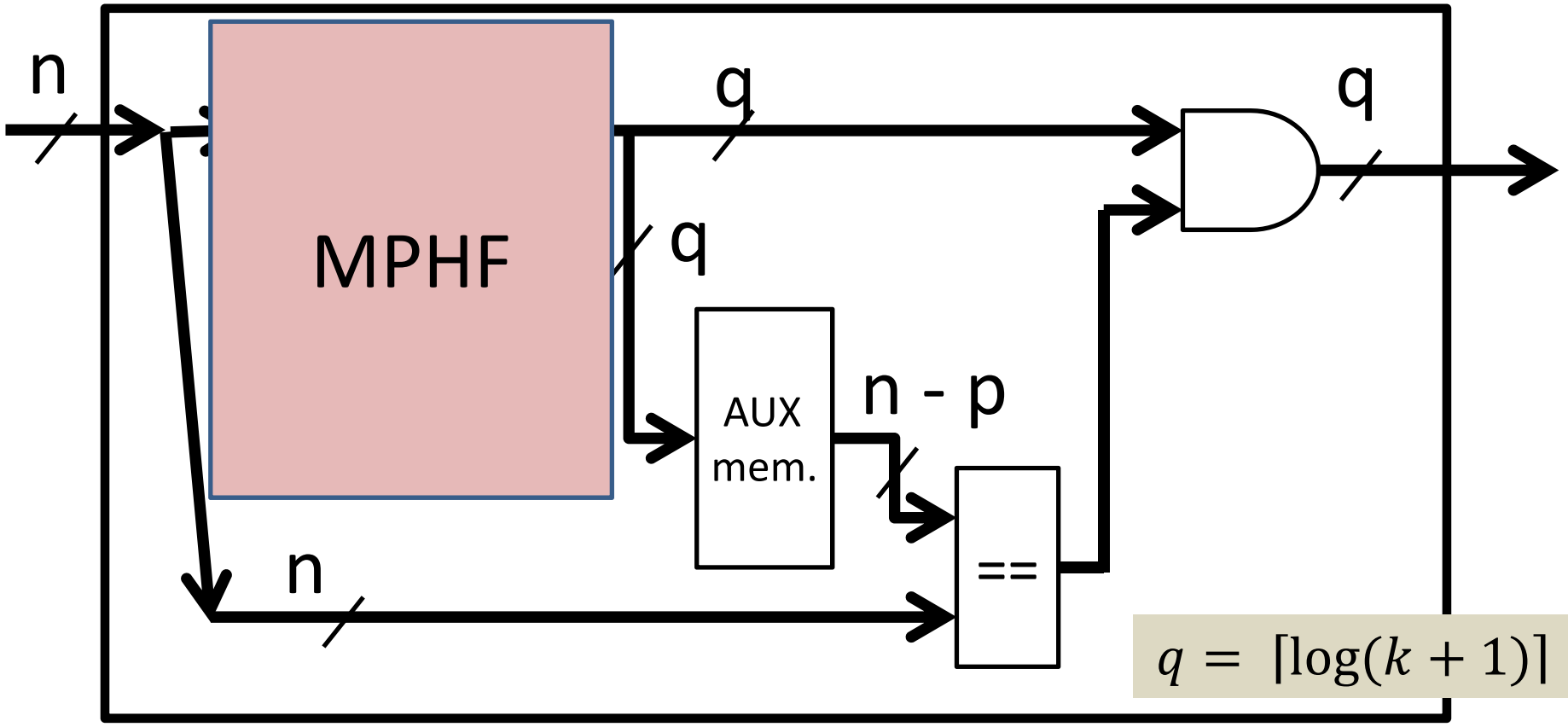
AUX単体と並列インデックス生成器のメモリサイズ

$p \leq q$ なら Main メモリと AUX メモリの和が通常の AUX メモリよりも小さくなる。



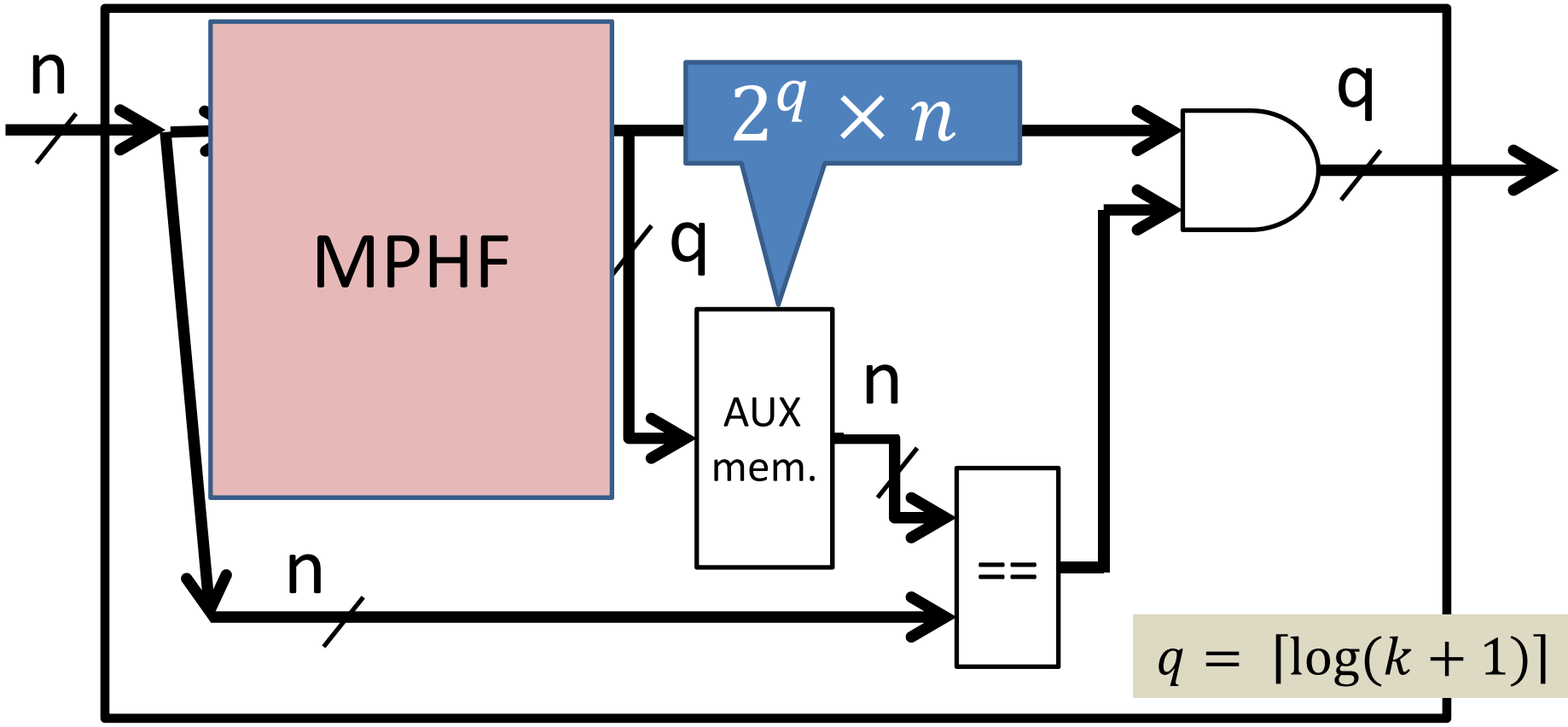
AUXメモリのオーバーヘッド

- MPHF以外に常にAUXメモリのオーバーヘッドが存在
- 行シフト分解も同様



AUXメモリのオーバーヘッド

- MPHF以外に常にAUXメモリのオーバーヘッドが存在
- 行シフト分解も同様



まとめ

- コンフリクトフリー分割に基づく並列インデックス生成器の構成法を提案
- トータルのメモリ量としてはほぼ理想的なサイズを実現
- 今後の課題:
  - 入力に偏りがある場合の入力ハッシュ関数の構成法(XORゲートを用いた線形変換)
  - ワイルドカードを含むパタンや区間に対するマッチング

ご清聴ありがとうございました。