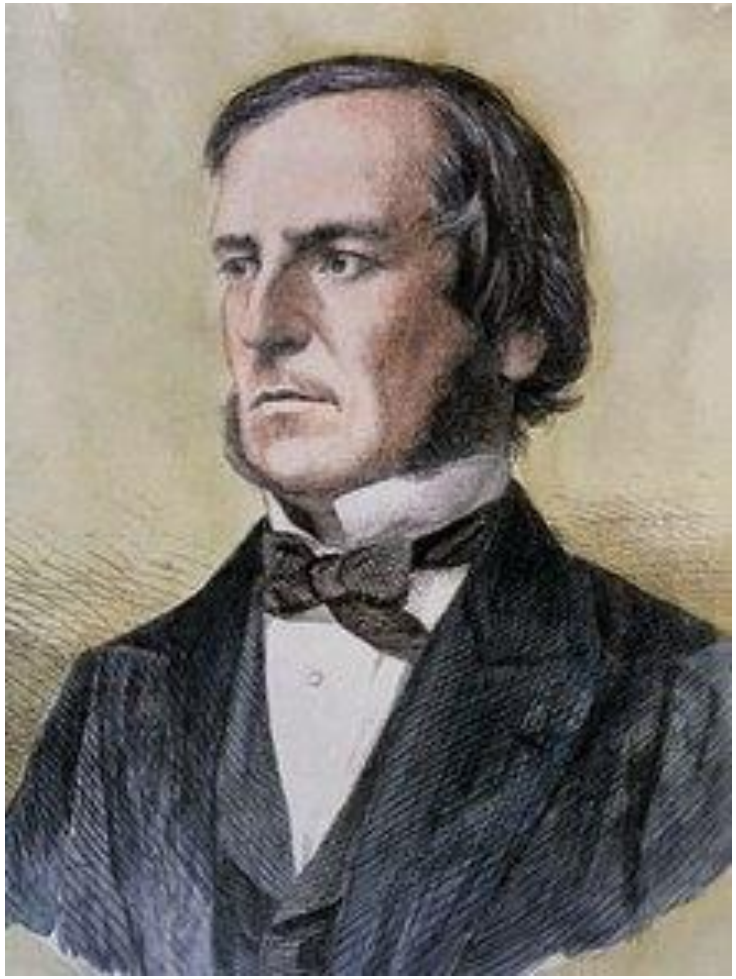
Булева алгебра, булеви функции, побитови операции и др.

Красимир Манев

MNKnowledge, 13.04.2020

Джордж Бул (1815 – 1864)



Английски философ и математик.

Разработва математическа теория с която **формализира** неформалната до момента философска Логика и поставя началото на Математическата логика

Съждително смятане

Деф. а) ***Съждение*** е всяко твърдение, което можем да оценим с една от двете оценки „вярно“ (true) или „невярно“(false).

б) Нека p и q са съждения.

в) Тогава съждения са и:

- p ***или*** q – вярно, ако поне едно от двете е вярно;
- p ***и*** q – вярно, ако и двете са верни;
- ***не е вярно*** p – вярно, ако p не е вярно.

Такъв формален модел напълно съответства на обичайната човешка логика и на начина по който провеждаме доста математически разсъждения.

Булева алгебра (БА)

Да разгледаме простото множество $\mathcal{B} = \{0,1\}$ където 1 съответства на логическото „вярно“ (true) а 0 на логическото „невярно“ (false).

Нека $x,y \in \mathcal{B}$. Дефинираме двуаргументните операции

- ***дизюнкция*** (съответстваща на ***или***) със знак $\vee$
- ***конюнкция*** (съответстваща на ***и***) със знак $\wedge$
- **отрицание** (съответстваща на ***не е вярно че***) със знак $\neg$

x	y	$\vee$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$\wedge$
0	0	0
0	1	0
1	0	0
1	1	1

x	$\neg$
0	1
1	0

Такава алгебрична структура наирчаме ***булева алгебра*** – $(\mathcal{B}; \vee, \wedge, \neg)$.

Булева алгебра - свойства

- Комутативни: $x \vee y = y \vee x;\ \ x \wedge y = y \wedge x$
- Асоциативни: $(x \vee y) \vee z = x \vee (y \vee z) = x \vee y \vee z;$

  $(x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y \wedge z$
- Дистрибутивни: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z);$

  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- Идемпотентни: $x \vee x = x;\ x \wedge x = x$
- Константи: $x \vee 1 = 1;\ x \vee 0 = x;\ x \wedge 1 = x;\ x \wedge 0 = 0;$
- Отрицание: $x \vee \neg x = 1;\ x \wedge \neg x = 0;\ \neg(\neg x) = x$
- Закони на Де Морган: $\neg(x \vee y) = \neg x \wedge \neg y$

  $\neg(x \wedge y) = \neg x \vee \neg y$

Булева алгебра - свойства

Всяко от гореизброените свойства може лесно да се докаже, като се построят ***таблиците на истинност*** на изразите от двете страни на равенството. Да докажем така първия закон на Де Морган $\neg(x \vee y) = \neg x \wedge \neg y$

x	y	$x \vee y$	$\neg(x \vee y)$		$\neg x \wedge \neg y$	$\neg x$	$\neg y$
0	0	0	1	=	1	1	1
0	1	1	0	=	0	1	0
1	0	1	0	=	0	0	1
1	1	1	0	=	0	0	0

Задача. Докажете едно от дистрибутивните свойства.

Указание. Тъй като в него участват три променливи, таблиците на истинност ще имат по 8 реда!!!

Булева алгебра - свойства

Интересно свойство във всяка БА ни дава следната

Теорема (дуалност на БА) . Ако в твърдение τ в БА заменим конюнкция с дизюнкция и обратно, получаваме твърдение τ' със същата вярност като тази на τ.

Например, ако направим описаната замяна в първия закон на Де Морган

$$\neg(x \vee y) = \neg x \wedge \neg y \Rightarrow \neg(x \wedge y) = \neg x \vee \neg y$$

получаваме втория закан на Де Морган.
Аналогично, от комутативността и асоциативността на дизюнкцията - комутативността и асоциативността на конюнкцията и т.н.

Булева алгебра на множествата

БА на верността на съждения не е единствен пример за БА. Да разгледаме някакво универсално множество $\mathcal{U}$ и фамилията от неговите подмножества $2^{\mathcal{U}}$, с обичайните операции:

- ***обединение*** (съответстваща на $\vee$) със знак $\cup$
- ***конюнкция*** (съответстваща на $\wedge$) със знак $\cap$
- **допълнение до** $\mathcal{U}$ (съответстваща на $\neg$) със знак $\neg$

$x \in A$	$y \in B$	$x \in A \cup B$
0	0	0
0	1	1
1	0	1
1	1	1

$x \in A$	$y \in B$	$x \in A \cap B$
0	0	0
0	1	0
1	0	0
1	1	1

$x \in A$	$x \in \neg A$
0	1
1	0

Очевидно $(2^{\mathcal{U}}; \cup, \cap, \neg)$ е също булева алгебра със същите свойства. Свойствата са същите във всяка БА!!!

Булева алгебра на булевите функции

Нека $\mathcal{B} = \{0,1\}$, а $\mathcal{B}^n = \{(a_1,a_2,\ldots a_n) \mid a_i \in \mathcal{B}\}$ е множеството от *n*-мерните двоични вектори.

Деф. Функциите от $\mathcal{F}_n = \{f: \mathcal{B}^n \to \mathcal{B} \mid n = 1, 2, \ldots\}$ наричаме ***булеви*** или ***двоични функции (БФ).*** Тъй като $|\mathcal{B}^n| = 2^n$, броят на функциите $|\mathcal{F}_n| = 2^{2^n}$. Функциите на 1 променлива са 4, а на 2 променливи – 16. Добре е да различаваме булевата стойност 0 (скалар) от булевата функция **0** (вектор)

x	**0**	x	$\neg x$	**1**
0	0	0	1	1
1	0	1	0	1

x	y	**0**	$\wedge$		x		y	$\oplus$	$\vee$			$\neg y$		$\neg x$			**1**
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Булева алгебра на булевите функции

Както се вижда от двете таблици:

- Всяка БФ на n-1променливи се среща и като функция на n променливи, но с двойно по-дълъг стълб от стойности
- Функциите, чиито стойности повтарят стойността на някоя от променливите $f(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i$ наричаме ***идентитети***. Всяко добавяне на променлива, добавя нов идентитет и отрицание на новата променлива.
- Една от функциите на две променливи, която я няма при тези на една променлива е $f(x,y) = x \oplus y$, която наричаме ***събиране*** (***по модул 2***).
- Всяка функция на $k<n$ променливи можем да разширим до функция на n променливи с добавяне на нови (т.н. ***фиктивни***) променливи

Булева алгебра на булевите функции

Да разгледаме множеството $\mathcal{F}_n$ на БФ на *n* променливи.
Дефинираме следните ***побитови операции***:

$f(x_1,\ldots,x_n)$ **|** $g(x_1,\ldots,x_n) = h(x_1,\ldots,x_n)$ такава, че

$h(a_1,\ldots,a_n) = f(a_1,\ldots,a_n) \vee g(a_1,\ldots,a_n),\ \forall (a_1,\ldots,a_n) \in \mathcal{B}^n$

- (***побитова***) ***дизюнкция*** на f и g

$f(x_1,\ldots,x_n)$ **&** $g(x_1,\ldots,x_n) = h(x_1,\ldots,x_n)$ такава, че

$h(a_1,\ldots,a_n) = f(a_1,\ldots,a_n) \wedge g(a_1,\ldots,a_n),\ \forall (a_1,\ldots,a_n) \in \mathcal{B}^n$

- (***побитова***) ***конюнкция*** на f и g

! $f(x_1,\ldots,x_n) = h(x_1,\ldots,x_n)$ така, че $h(a_1,\ldots,a_n) = \neg f(a_1,\ldots,a_n)$
$\forall (a_1,\ldots,a_n) \in \mathcal{B}^n$ **-** (***побитово***) ***отрицание*** на f

$f(x_1,\ldots,x_n)$ **^** $g(x_1,\ldots,x_n) = h(x_1,\ldots,x_n)$ такава, че

$h(a_1,\ldots,a_n) = f(a_1,\ldots,a_n) \oplus g(a_1,\ldots,a_n), \forall (a_1,\ldots,a_n) \in \mathcal{B}^n$

- (***побитово***) ***събиране по модул 2*** на f и g

($\mathcal{F}_n$; **|** , **&** , **!**) също е булева алгебра

Булева алгебра на булевите функции

Деф. Множеството от БФ C наричаме ***пълно***, ако всяка БФ може да се представи с формула над C. По традиция, вместо **|** ще използваме $\vee$ а вместо $\&$ - нищо.

Нека да означим с $x^1 = x$, а с $x^0 = !x$. Функция от вида $x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n}$ наричаме ***елементарна конюнкция.***

Лема. $x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n} = 1$ тстк $x_i = \sigma_i$, $i = 1, 2, \ldots, n$

Теорема. Множеството БФ { **|**, &, **!** } е пълно.

Доказателство: Нека $f(x_1, \ldots, x_n)$ е БФ

а) ако $f(x_1, \ldots, x_n) = \mathbf{0}$, тогава $f(x_1, \ldots, x_n) = x_1$ **&** $(!x_1)$;

б) нека $f(x_1, \ldots, x_n) \neq \mathbf{0}$.

Сега $f(x_1, \ldots, xn) = \bigvee_{\forall f(\sigma_1, \ldots, \sigma_n)=1} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n}$,

Булева алгебра на булевите функции

$$f(x_1,\ldots,xn)= \bigvee_{\forall f(\sigma_1,\ldots,\sigma_n)=1} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n},$$

т.е. побитова дизюнкция на т.н. ***елементарни конюнкции***, всяка от които съответства на единична стойност на функцията, а степените на променливите са съответните стойности във вектора за който f е единица - ***съвършена дизюнктивна нормална форма*** (СДНФ)

Ако $f(a_1,\ldots,a_n) = 1$ тогава $\bigvee_{\forall f(\sigma_1,\ldots,\sigma_n)=1} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n}$ $= a_1^{a_1} a_2^{a_2} \ldots a_n^{a_n} = 1$ също, а ако

$f(a_1,\ldots,a_n) = 0$ тогава $\bigvee_{\forall f(\sigma_1,\ldots,\sigma_n)=1} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n} = 0$

защото $(a_1,\ldots,a_n)$ не съвпада с никой $(\sigma_1,\ldots,\sigma_n)$

Булева алгебра на булевите функции

Пример. Нека напишем СДНФ на побитовата дизюнкция.

Тъй като векторите върху които тя е единица са $(0,1)$, $(1,0)$ и $(1,1)$ получаваме

$$x \vee y = x^0y^1 \vee x^1y^0 \vee x^1y^1$$

или според традициите да се поставя отрицанието като черта над буквата

$$x \vee y = \bar{x}y \vee x\bar{y} \vee xy$$

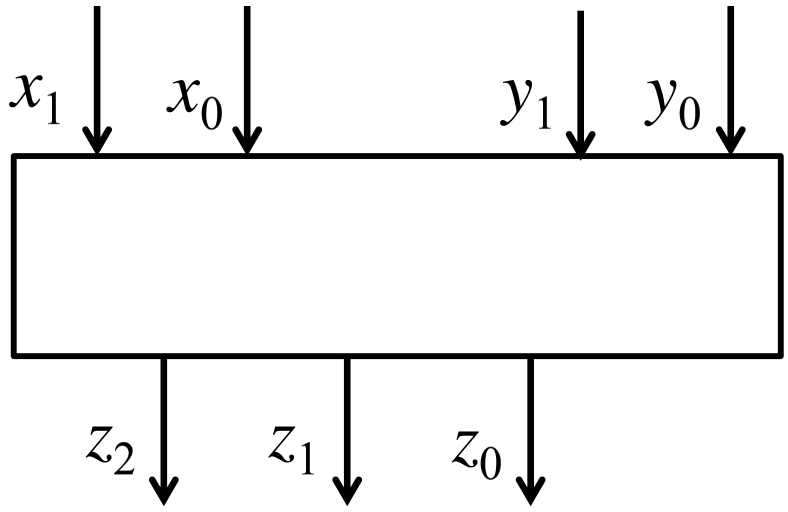
Задача. Напишете СДНФ на функцията от таблицата вдясно.

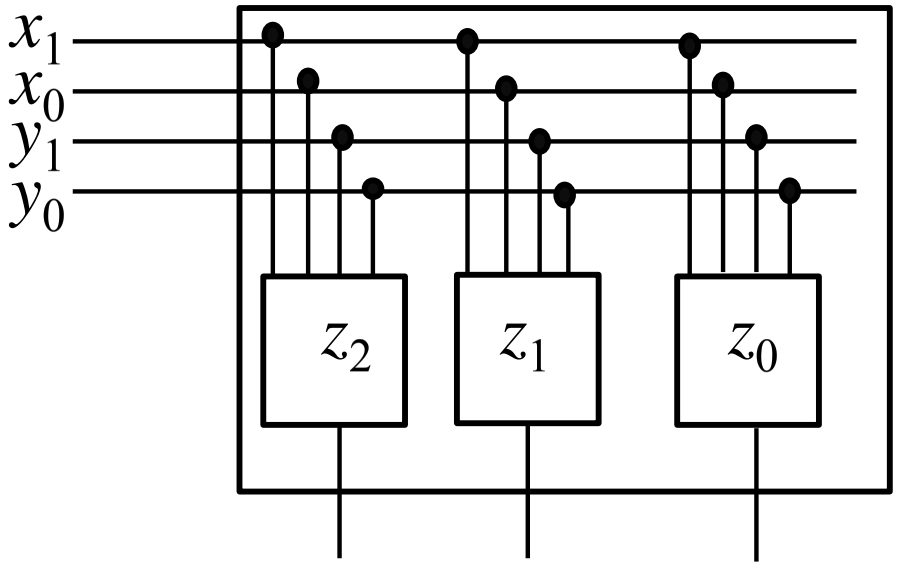
x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Както се вижда, всяка функция, без $\mathbf{0}$, може да има повече от една ДНФ но точно една СДНФ. Това поражда задачата за търсене на ***минимална ДНФ*** на фунцията.

Ролята на БФ – схеми от ФЕ

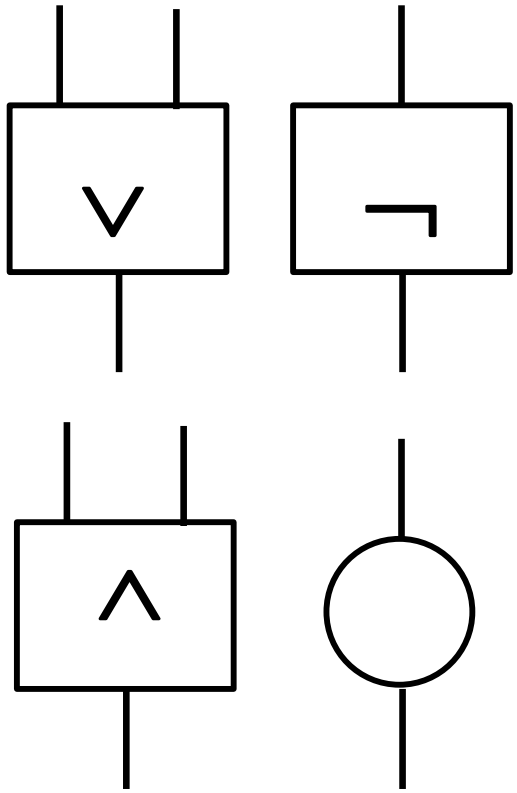
На Фиг. вляво е показано просто устройство – 2-разряден суматор. Общият случай (n-разряден суматор) не се различава съществено. То пресмята булевите функции $z_2(x_1,x_0,y_1,y_0)$, $z_1(x_1,x_0,y_1,y_0)$ и $z_0(x_1,x_0,y_1,y_0)$

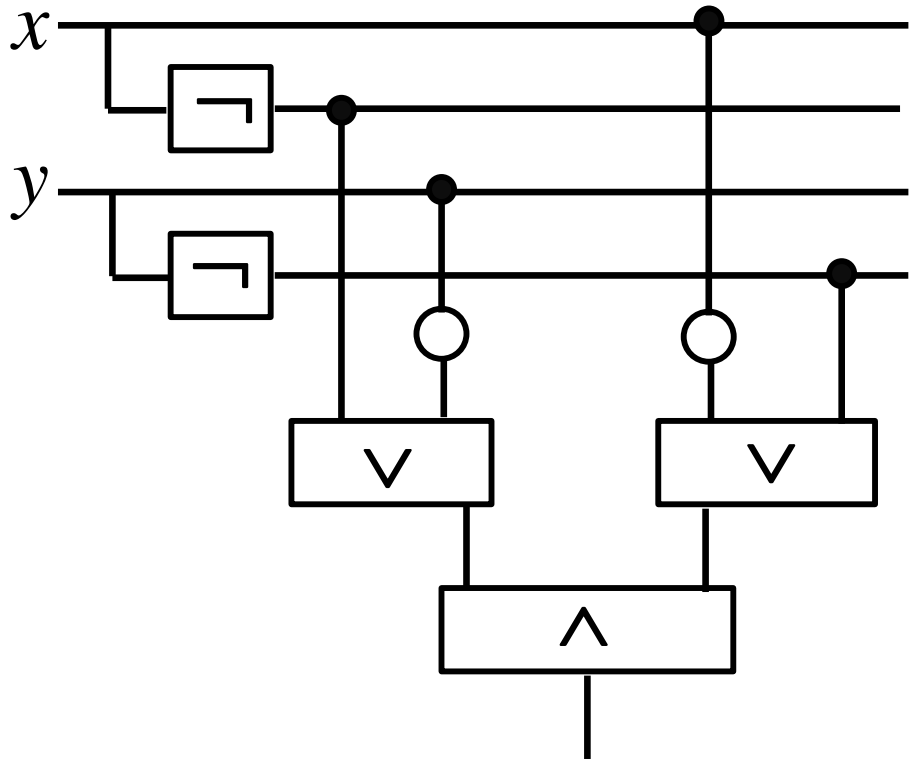




Ролята на БФ – схеми от ФЕ

На Фиг. вляво са показани т.н. ***функционални елементи*** (ФЕ) - пълно множество. По СДНФ на събирането по модул 2 $x \oplus y = \bar{x}y \vee x\bar{y}$ строим ***схема от ФЕ***, показана на Фиг. вдясно





Език за програмиране

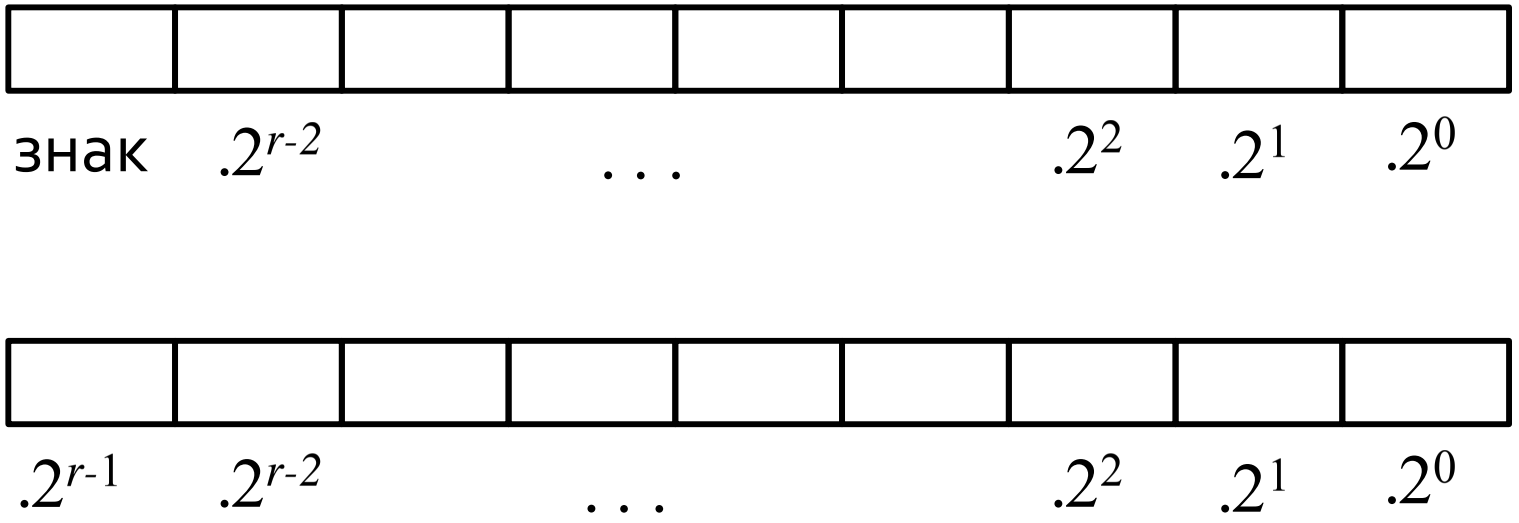
В езиците за програмиране използваме две форми на БА: а) булевата логика

- Операциите за сравнение {<, >, ==, <=, >=, !=} пораждат булевите стойности **true (1)** и **false (0)**
- Цялото 0 е еквивалентно на **false**
- Всяко цяло ≠ 0 е еквивалентно на **true**
- Знакът за дизюнкция е ||, знакът за конюнкция е **&&**, а знакът за отрицание е !
- Логически изрази (условия)

```
ch>='a'&&ch<='z'||ch>='A'&&ch<='Z`
!(ch>='0'&&ch<='9')
```

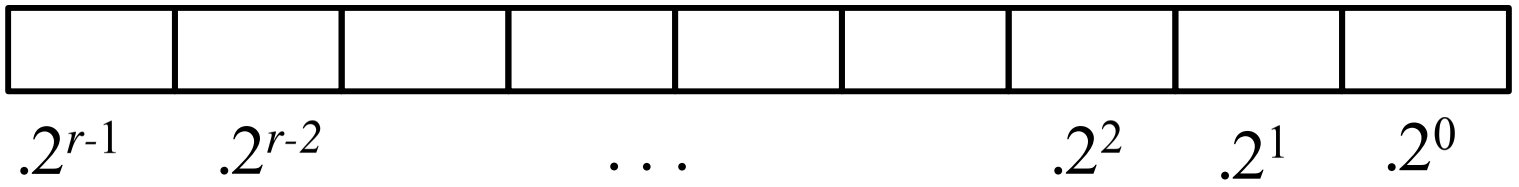
Език за програмиране

б) Другата форма на използване на БА в ЕП (примерите са за С) е БА на побитовите операции. В обичайните компютърни архитектури има две групи целочислени типове – знакови и беззнакови (**`unsigned`**)

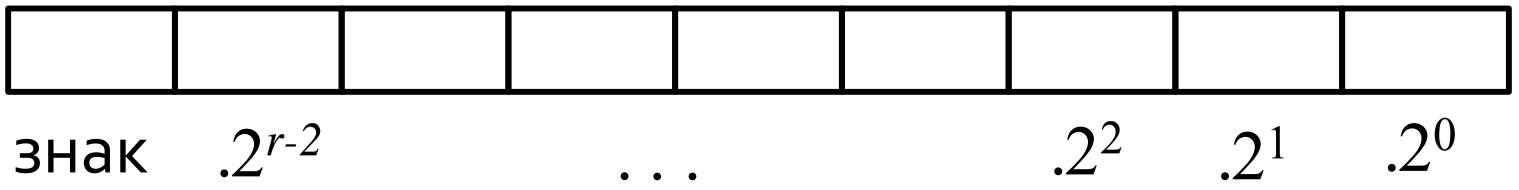


Език за програмиране

В r-разрядния беззнаков тип се представят, в двоична бройна система, целите неотрицателни числа от 0 до 2^r-1

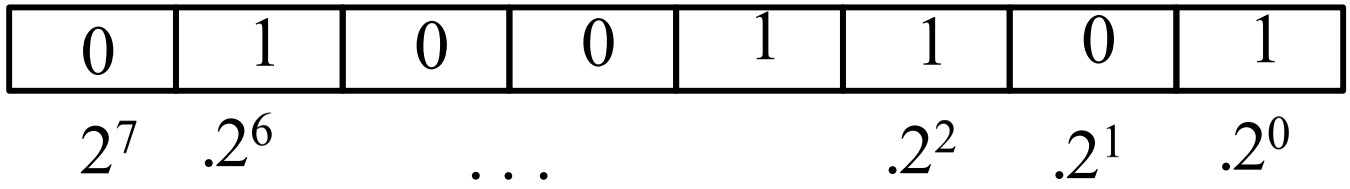


В r-разрядния знаков тип се представят целите числа от -2^{r-1} до $2^{r-1}-1$.



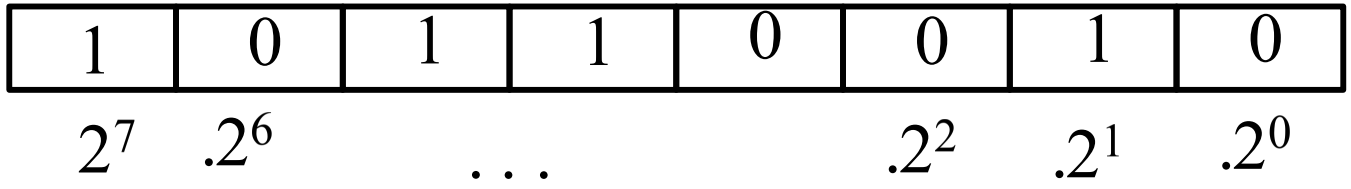
Език за програмиране

В r-разрядния знаков тип неотрицателните числа от 0 до $2^{r-1}-1$ се представят, в двоична бройна система, като знаковият бит е 0

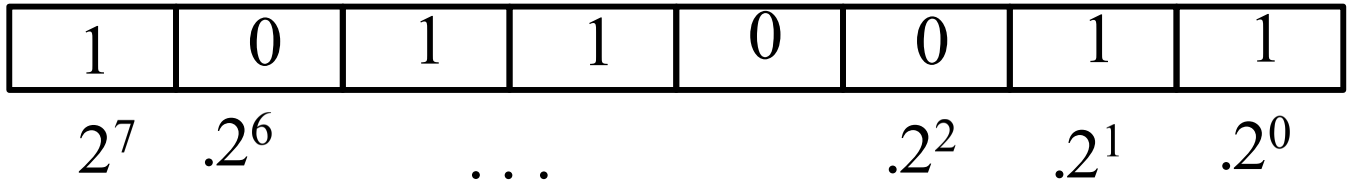


Съответното отрицателно се представя в ***допълнителен код***

invert



+1



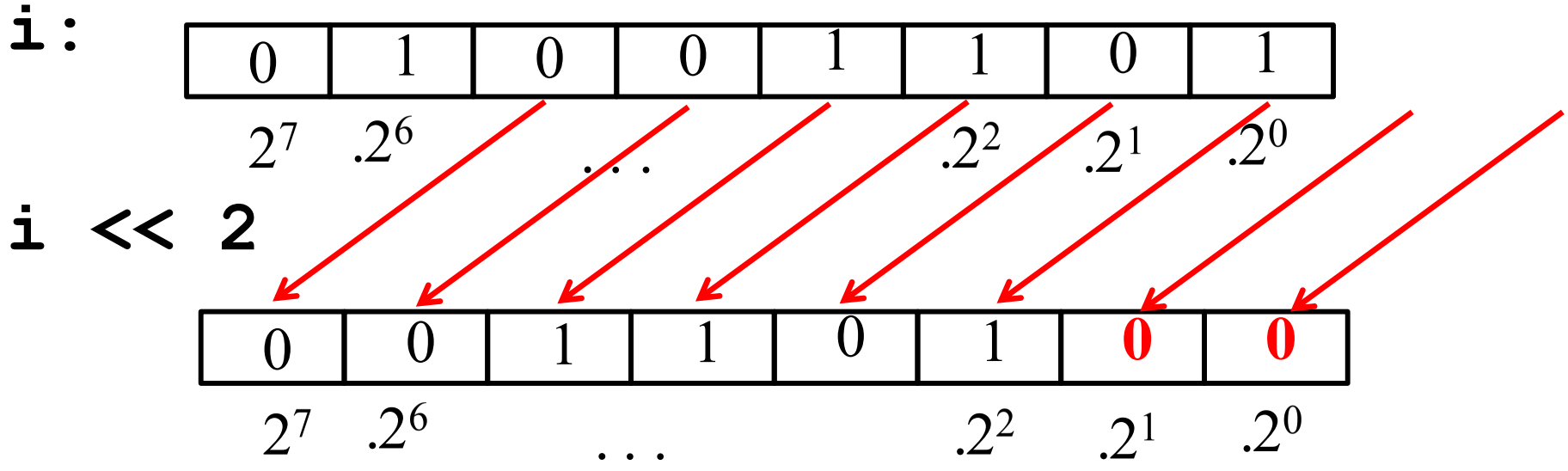
Език за програмиране

В езика С са допустими няколко побитови операции (**`unsigned i,k;`**):

- ***Побитово изместване наляво беззнаково***

`i << k`

Съдържанието на **`i`** се измества **`k`** бита наляво

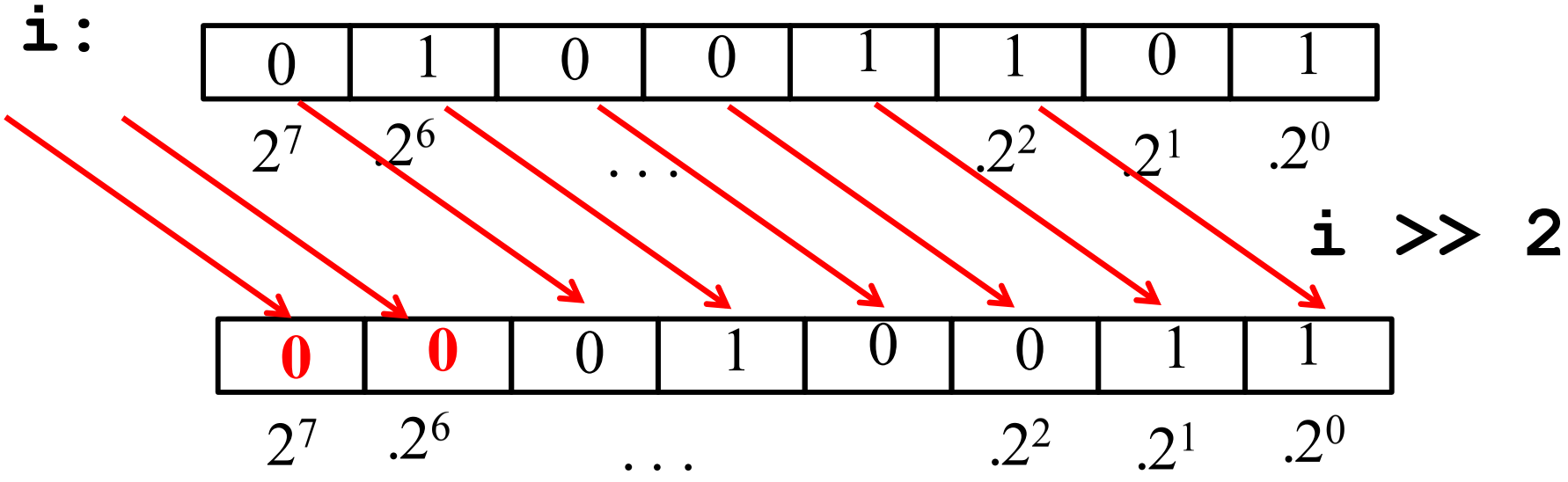


Език за програмиране

- ***Побитово изместване надясно беззнаково*** (**`unsigned i,k`**)

`i << k`

Съдържанието на **`i`** се измества **`k`** бита надясно



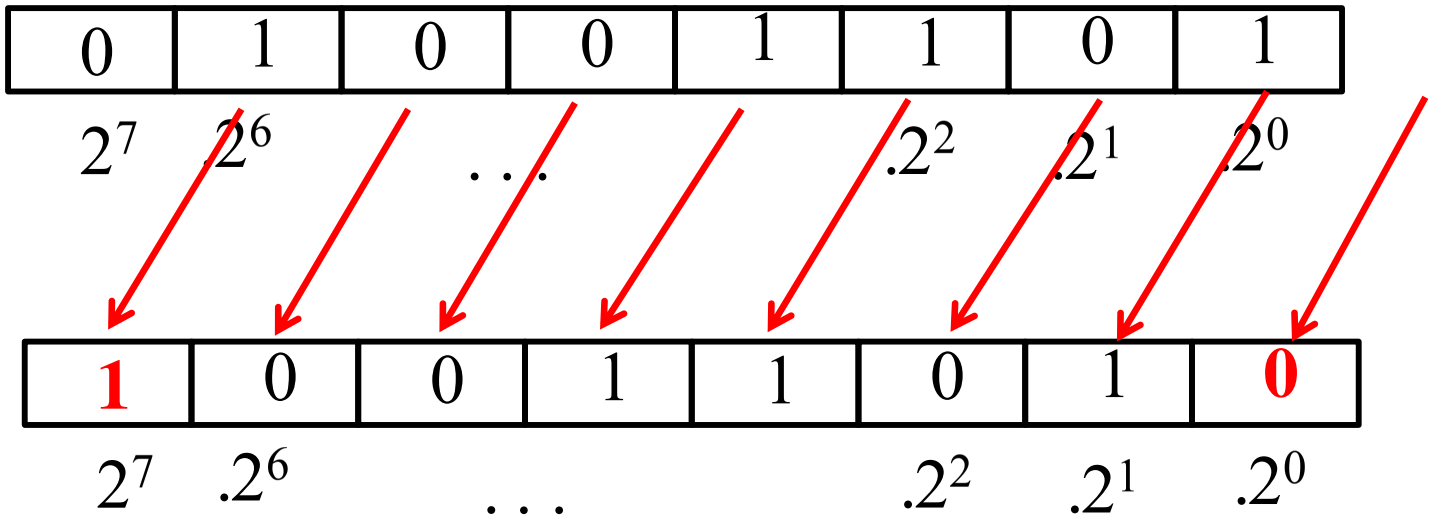
Език за програмиране

Знаковото изместване има специфика:

- ***Побитово изместване наляво знаково***

При знаково изместване наляво „настъпващите“ от дясно битове влизат като знакови и полученото число е ту положително, ту отрицателно (**`int i; unsigned k`**)

`i:`

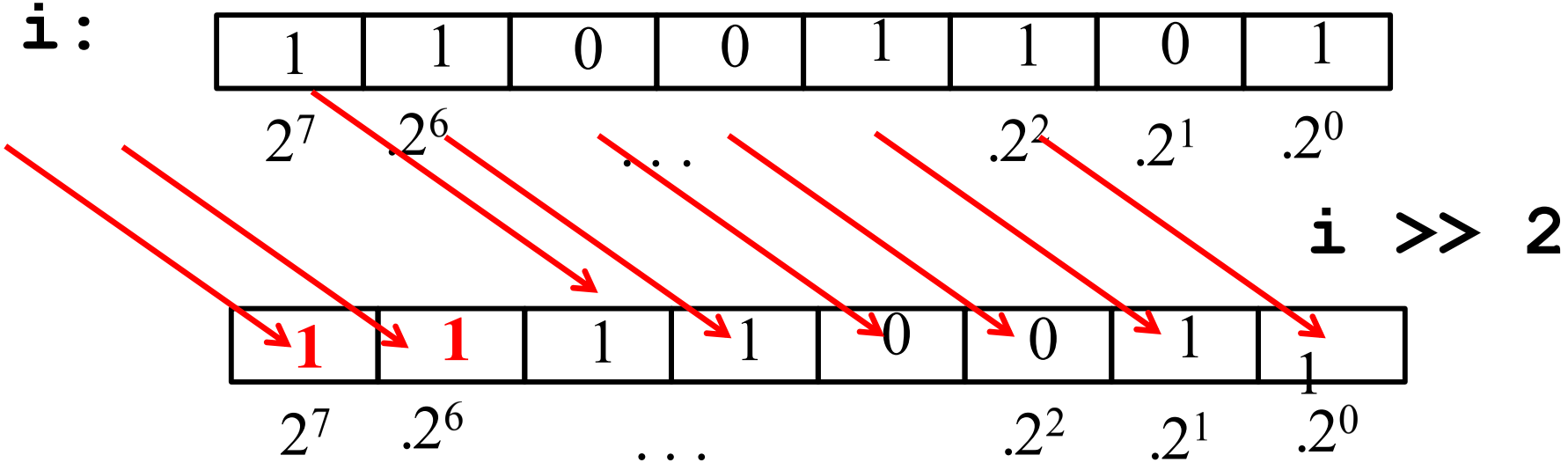


Език за програмиране

- ***Побитово изместване надясно знаково*** (**`int i; unsigned k;`**)

При знаковото изместване надясно, попълването на освобождаваните позиции става със знаковия бит!!!

`i:`



Език за програмиране

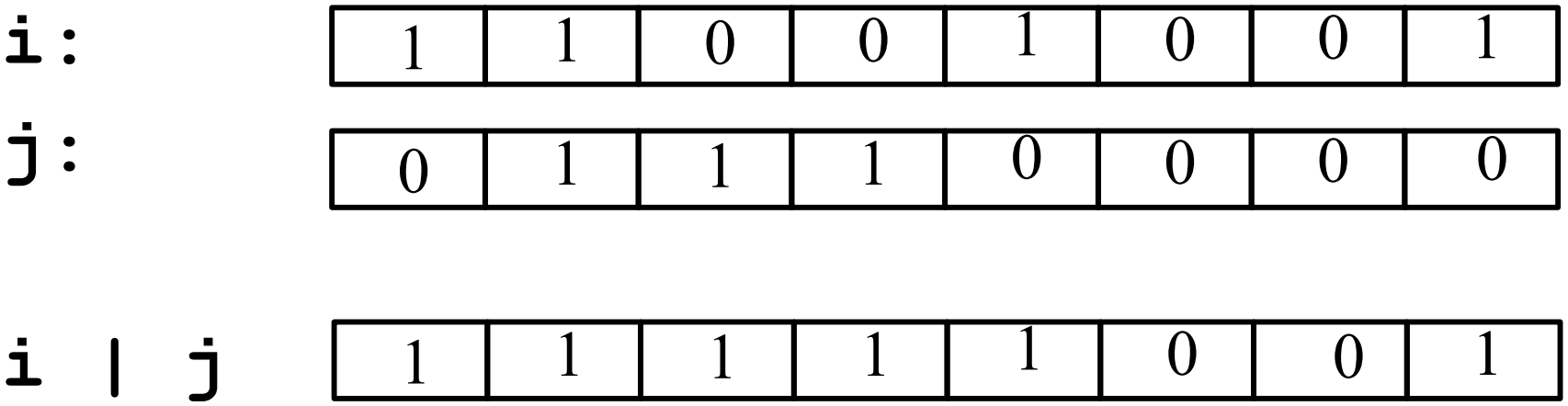
- Както се вижда, побитовите операции имат особености при знаковите целочислени типаве и затова препоръчвам да се използват само при беззнаковите
- По принцип С компилаторите не отказват да компилират измествания с отрицателен знак (`i<<-1`) но резултатът е трудно презвидим дори за опитен програмист. Затова препоръчваме да не се използват отрицателни измествания

Ролята на БФ – Език за програмиране

- ***Побитова дизюнкция*** **`(unsigned i,j;)`**

```
i | j
```

При побитовата дизюнкция всеки бит в резултата е дизюнкция на съответните битове на аргументите

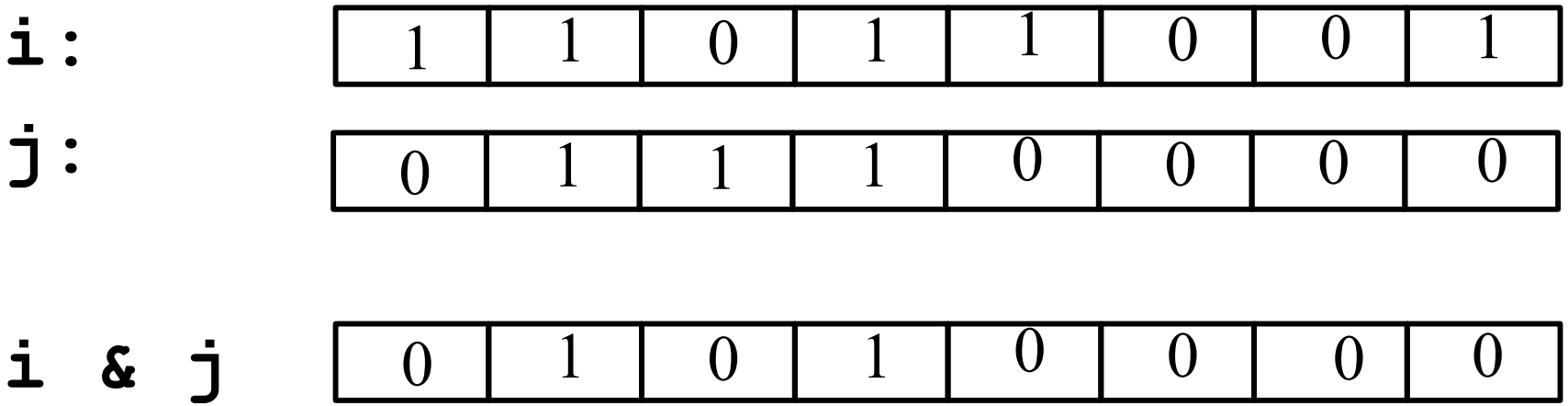


Ролята на БФ – Език за програмиране

- ***Побитова конюнкция*** **`(unsigned i,j;)`**

  `i & j`

При побитовата конюнкция всеки бит в резултата е конюнкция на съответните битове на аргументите

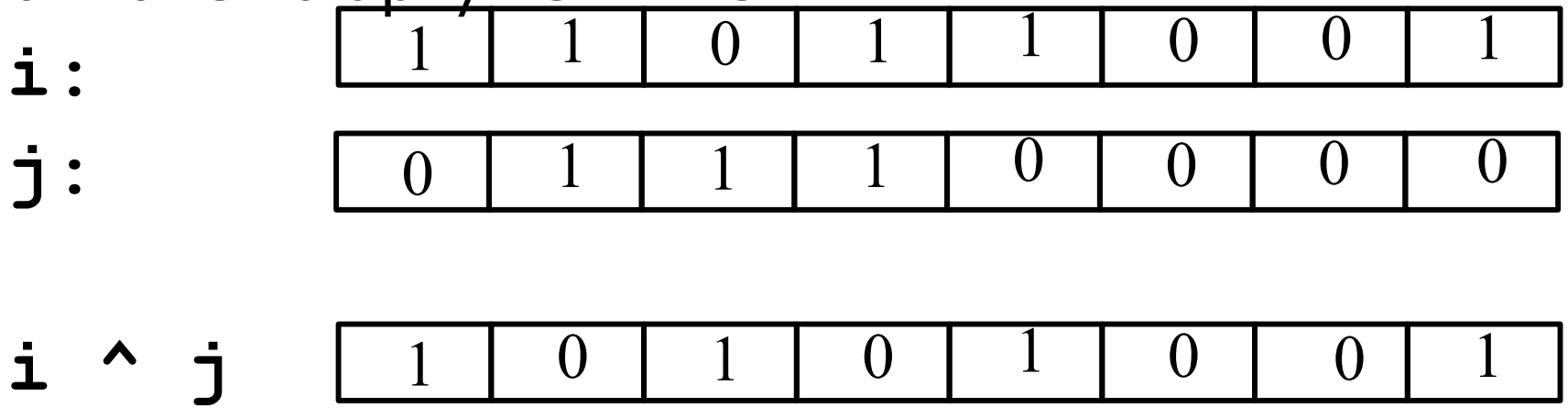


Ролята на БФ – Език за програмиране

- ***Побитово събиране по модул 2*** **`(unsigned i,j;)`**

```
i ^ j
```

При побитовото събиране по модул 2 всеки бит в резултата е събиране по модул 2 на съответните битове на аргументите



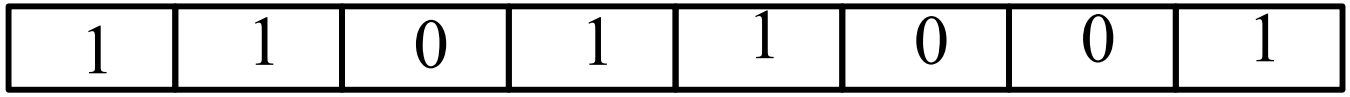
Ролята на БФ – Език за програмиране

- ***Побитово отрицание*** **`(unsigned i,j;)`**

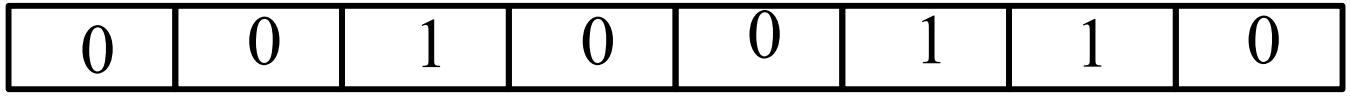
  `~i`

При побитовото събиране по модул 2 всеки бит в резултата е събиране по модул 2 на съответните битове на аргументите

`i:`



`~i:`



Използване на побитови операции

Лема. Очевидно `i^i = 0`

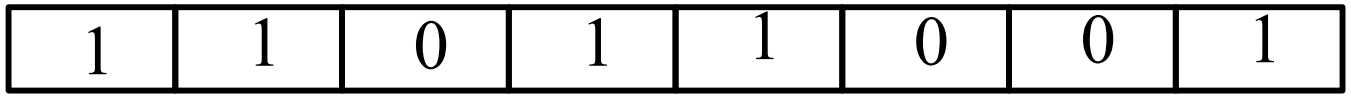
Нека имаме `unsigned i,j;`

Операция	i	j
`i^=j;`	`i^j`	`j`
`j^=i;`	`i^j`	`i`
`i^=j;`	`j`	`i`

Значи кодът `i^=j; j^=i; i^=j;`
Размества съдържанието на `i` и `j` без използване на друга променлива

Използване на побитови операции

Нека $A = \{0,1,2,\ldots,N\text{-}1\}$, а $M = \lceil N/(8*\text{sizeof(type)}) \rceil$. Тогава всяко подмножество A можем да представим в **`<type> a[M];`** като редица от N бита, в която i-тият по ред бит 0, ако $i \notin A$ и е 1, ако $i \in A$. Например, $\{0,1,3,4,7\} \subseteq \{0,1,\ldots,7\}$ представяме с



Използване на побитови операции

- При това представяне можем да реализираме **много по-бързо**, отколкото при класическо представяне със списък на елементите, операциите с множества
- При това представяне използваме 8*sizeof(type) пъти **по-малко памет**, което ни позволява да представяме подмножествата на много по-големи множества

Използване на побитови операции

Ако type **a[M],b[M];** са две подмножества

обединение

```
for(int i=0;i<M;i++) a[i]|=b[i];
```

сечение

```
for(int i=0;i<M;i++) a[i]&=b[i];
```

симетрична разлика

```
for(int i=0;i<M;i++) a[i]^=b[i];
```

Допълнение до универсалното

```
for(int i=0;i<M;i++) a[i]=~a[i];
```

Използване на побитови операции

За да можем да работим с такова представяне на множества са ни нужни още няколко основни операции:

- *Добавяне на елемент K (void set(K))*
- *Премахване на елемент K (void unset(K))*
- *Проверка дали елемент K принадлежи на множеството (int check(K))*

За целта **`K/(8*sizeof(type))`** определя елемента на масива в който се съдържа елементът **`K`**, а **`K%(8*sizeof(type))`** позицията му (отляво надясно в елемента)

Използване на побитови операции

Нека **type** за по-поросто е **int**

```
int a[],M;
void set(unsigned K)
{ int i = K/32,j = K%32;
  unsigned p = 1<<(31-j);
  a[i]| = p;
}
```

Използване на побитови операции

```
int a[],M;
void unset(unsigned K)
{ int i = K/32,j = K%32;
  unsigned p = ~(1<<(31-j));
  a[i]& = p;
}
```

Използване на побитови операции

```
int a[],M;
int check(unsigned K)
{ int i = K/32,j = K%32;
  unsigned p = 1<<(31-j);
  unsigned b=a[i]&p;
  return b;
}
```

Използване на побитови операции

Задача. Напишете програми, които по зададени две множества от цели положителни числа по малки от 10^6 с M и N елемента

proga) намира и извежда сечението им;

progb) намира и извежда разликата им – т.е. множеството от елементите на първото, които не привнадлежат на второто;

progc) изтрива от двете изтрива от двете множества елементите на сечението им.

Използване на побитови операции

Задача. Една често срещана в комбинаторните изследвания задача е да се намери броят на елементите на зададено множество.

При представянето по-горе това можем да направим с кода:

```
int count  = 0, i, a[],N;
for (i=0;i<N;i++)
    if(check(i)) count++;
```

В този случай сложността на алгоритъма е $O(N)$.Съществува обаче и по-бърз алгоритъм който също използва побитова операция

Използване на побитови операции

```
int count = 0,N;unsigned a[];
int size()
{int M = ceiling(N);
 for (i=0;i<M;i++)
 { while(a[i]!=0)
    {a[i]&=a[i-1]; count++;}
 return count;
}
```

Обяснение. Операцията **x&(x-1)** заменя най-дясната единица в **x** с нула. Например

```
100100&100011 = 100000
100000&011111 = 0
```

Използване на побитови операции

- В състезателното програмиране е много полезна алгоритмичната схема ***Динамично програмиране*** (или ***оптимиране***).
- Същността на ДП е да се свежда решението на задача с даден размер на входа към решенията на същата задача при по-малки размери на входа
- Решението в стил ДП използва **таблица**, в която се съхраняват решенията на различните подзадачи и тези решения се използват наготово.
- Възможностите на ДП да реши задача с голям размер зависи от това **колко голяма таблица може да се разположи в паметта**.