Семинар 6. Общие принципы вычислений

6.1 Основные понятия алгебры логики

Алгебра логики — раздел математической логики, изучающий **высказывания**, рассматриваемые со стороны их логических (истина или ложь) значений, и логические операции над ними.

Алгебра логики возникла в середине 19 века в работах Дж. Буля и была развита Ч. Пирсом (C.S. Peirs), П.С. Порецким, Б. Расселом (B. Russel), Д. Гильбертом (D. Hilbert) и др. **Высказыванием называются предложения, которые могут быть охарактеризованы понятием — истина или ложь**. Использование логических связок "и", "или", "если. . . то", "эквивалентно", частица "не" и т.д. позволяет строить новые, более сложные, высказывания из заданных. Истинность или ложность сложных высказываний зависит от истинности или ложности исходных высказываний. Для обозначения истинности вводятся тождественные символы:

$$\text{Истина} \equiv \text{И} \equiv \text{True} \equiv \text{T} \equiv 1. \tag{6.1}$$

Для обозначения ложности высказывания вводятся следующие тождественные символы

$$\text{Ложь} \equiv \text{Л} \equiv \text{False} \equiv \text{F} \equiv 0. \tag{6.2}$$

Соответственно для логических связок приняты следующие обозначения:

$$\left.\begin{array}{l} \text{"и" (конъюнкция)} \equiv \& \equiv \wedge \equiv \text{AND} \equiv \cap \\ \text{"или" (дизъюнкция)} \equiv \vee \equiv \text{OR} \\ \text{"если . . . то" (импликация)} \equiv \rightarrow \\ \text{"эквивалентность"} \equiv \sim \\ \text{"отрицание"} \equiv \text{черта над высказыванием} \equiv \neg \equiv \text{NOT} \end{array}\right\} \tag{6.3}$$

Связки и частицы "не" рассматриваются в алгебре логики как операции над величинами, принимающие два значения 0 и 1, а высказывания с произвольными высказываниями и связками образуют формулы. При этом высказывания, образующие формулу, рассматриваются в качестве переменных, а связки в качестве функций. Формулы A и B называются равными $(A = B)$, если они реализуют равные функции.

Для задания функций алгебры логики, используются таблицы, содержащие все наборы значений переменных и значений функций: такие таблицы называются **таблицами истинности**. Пример таблицы истинности для NOT, AND, OR, импликации и эквивалентности приведены ниже

a	b	NOT a	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \sim b$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Сложные формулы в алгебре логики могут быть преобразованы. Для преобразования формул основную роль играют следующие законы:

– **закон коммутативности**

$$a \vee b = b \vee a; \qquad a \wedge b = b \wedge a; \tag{6.4}$$

– **закон ассоциативности**

$$(a \wedge b) \wedge c = a \wedge (b \wedge c); \qquad (a \vee b) \vee c = a \vee (b \vee c); \tag{6.5}$$

– **закон поглощения**

$$a \wedge (a \vee b) = a; \qquad a \vee (a \wedge b) = a; \tag{6.6}$$

– **закон дистрибутивности**

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge z); \qquad a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c); \tag{6.7}$$

– **закон противоречия**

$$a \wedge \overline{a} = 0; \tag{6.8}$$

– **закон исключения третьего**

$$a \vee \overline{a} = 1; \tag{6.9}$$

$$a \to b = \overline{a} \vee b; \qquad a \sim b = (a \wedge b) \vee (\overline{a} \wedge \overline{b}) \tag{6.10}$$

Множество всех формул, в построении которых учавствуют переменные высказывания, символы $\wedge$, $\vee$, $\to$, $\sim$, $\neg$, константы 0 и 1 называются **языком** над данными символами. Равенство (6.4)–(6.10) означают, что для всякой формулы в языке над $\wedge$, $\vee$, $\to$, $\sim$, $\neg$, 0, 1 найдется равная ей формула в языке над $\wedge$, $\vee$, $\neg$, 0, 1.

Алгебра логики развивалась под влиянием прикладных задач, среди которых приложение к теории электрических схем играет самое важное значение. В алгебре логики ставится задача минимизации функции приводя заданную функцию к функции имеющей наименьшее число сомножителей, то есть минимальную сложность. Такие функции называются минимальными.

В языке над $\wedge$, $\vee$, $\to$, $\sim$, 0, 1, $\oplus$, где $\oplus$ – используется для обозначения сложения по модулю 2 устанавливаются следующие соотношения:

$$a \vee b = ((a \wedge b) + a) + b \tag{6.11}$$

$$a \to b = \overline{a} \wedge b; \quad a \sim b = (a + b) + 1; \tag{6.12}$$

$$a + b = (a \wedge b) \vee (\overline{a} \wedge \overline{b}); \quad 1 = a \vee \overline{a}. \tag{6.13}$$

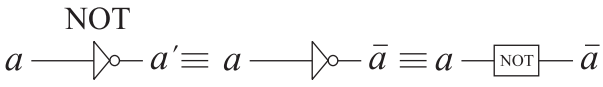
6.2 Классические универсальные машины и логические гейты

Универсальный компьютер — это логическое устройство, реализованное в виде сложной сети взаимосвязанных примитивных (основных) элементов. Для классического компьютера можно представить, что взаимосвязь элементов осуществляется идеальными проводниками, передающими одно из двух стандартных напряжений, представляющих локально один бит информации — 1 или 0. Сами примитивные элементы — или **гейты** реализуют функции преобразования, использующиеся в алгебре логики.

Классический компьютер осуществляет вычисление функций по заданным входным n-битам, располагая результат вычисления в m-битах. Функция с m-битами значений эквивалентна m-функциям, каждая из которых имеет однобитовое значение в качестве результата. Вычисление каждой из этих функций может быть сведено к последовательности элементарных логический операций (гейтов).

Символически гейты и биты, "соединенные проводами", изображаются рисунками.

Так на рисунке представлен примитивный элемент сети, в которой над битом выполняется логическая операция отрицания (NOT)

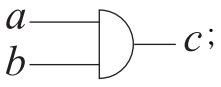


На рисунке указано, что бит a проходит через гейт NOT, который переворачивает бит, превращая 1 в 0 и 0 в 1. Линии до и после гейта NOT служат для переноса бита к гейту и удаление его после преобразования. Данные линии (провода) могут представлять как перенос бита из одной точки пространства в другую, так и развитие состояния бита во времени. Гейт NOT имеет один входной бит и один бит на выходе. Фактически, выходной бит вычисляет функцию $f(a) = 1 \oplus a$.

При построении сети предполагается, что сеть не содержит замкнутых петель.

Имеется много иных элементарных логических гейтов, полезных для организации процесса вычисления, которые имеют два бита в качестве исходных данных и один результирующий бит. Геометрические изображения, алгебраические формулы в бинарной арифметике и таблицы истинности, которых приведены ниже

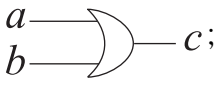
а. **AND − гейт**



$$c \equiv a \wedge b = a \cdot b$$

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

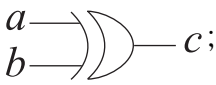
б. **OR-гейт**



$$c \equiv a \vee b = a + b - a \cdot b$$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

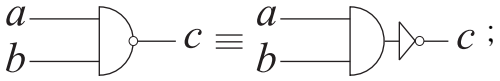
в. **XOR-гейт** (исключающее или ≡ "или", но не оба)



$$c \equiv a\,XOR\,b = a(1-b) + b(1-a)$$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

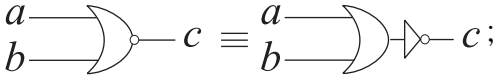
г. **NAND-гейт** ≡ (NOT AND-гейт)



$$c = 1 - a \cdot b$$

a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

д. **NOR-гейт** ≡ (NOT OR-гейт)



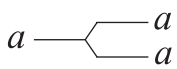
$$c = (1-a)(1-b)$$

a	b	c
0	0	1
0	1	0
1	0	0
1	1	0

Любое вычисление может быть записано в терминах булевского выражения, и любое булевское выражение может быть построено из фиксированного набора логических гейтов. Такой набор (например, AND, OR или NOT) называется универсальным. В действительности можно обойтись только двумя гейтами, такими как AND и NOT, или OR и NOT, или AND и XOR. Устройство, которое может исполнить произвольные комбинации логических гейтов из универсального набора, является универсальным компьютером.

Хотя приведенные выше гейты достаточны для математического аппарата алгебры логики, они недостаточны для реализации практической вычислительной машины. В реальном устройстве требуются еще два гейта FANOUT (разворачивание) и ERASE (стирание).

FANOUT



ERASE

FANOUT-гейт дублирует входной бит, а гейт ERASE — уничтожает входной бит. По сути FANOUT-гейт требуется для организации вычислений, а ERASE для очистки ячеек памяти компьютера.

В некоторых приложениях используется, помимо того, гейт EXCHANGE

EXCHANGE

a	b	a'	b'
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

Для примера рассмотрим цепь, которая суммирует два целых числа, имеющих длину n-бит. Базовым элементом в этой цепи является "ячейка" сети известная как полусумматор (half-adder $\equiv$ HA). На вход полусумматора подаются два бита x и y, а на выходе получается сумма битов $x \oplus y$ по модулю 2 и перенос (carry) бита в состоянии 1, если x и y оба 1, или 0 во всех остальных случаях.

Схема сети полусумматора имеет вид:

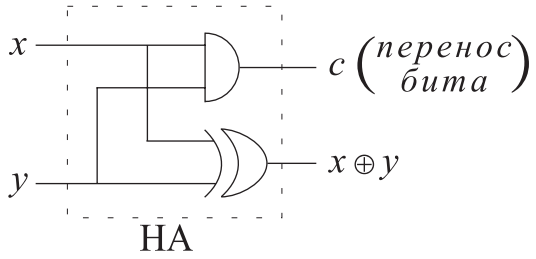
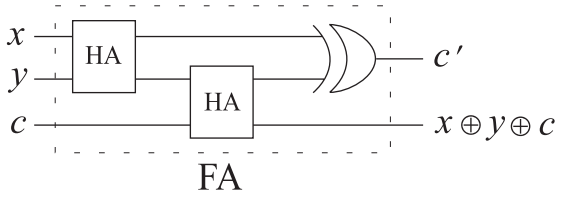


Таблица истинности полусумматора:

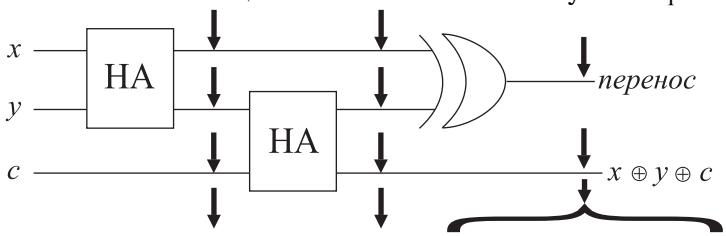
x	y	перенос	$x \oplus y$	двоичное число
0	0	0	0	00
0	1	0	1	01
1	0	0	1	01
1	1	1	0	10

Перенос бита позволяет перейти на следующий разряд, если складывается $1 + 1 = 0$. Каскад из 2-х полусумматоров (HA) образует полный сумматор (full-adder $\equiv$ FA)



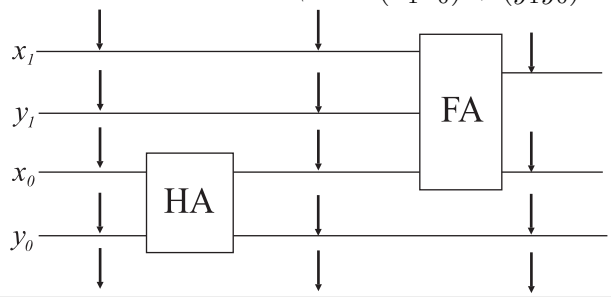
Полный сумматор имеет три бита на входе, где x, y—данные для сложения, c—перенос бита с предыдущего этапа вычислений и два бита на выходе. Один выходной бит является суммой по модулю 2 $x \oplus y \oplus c$ всех трех входящих битов, а второй выходной бит c'—есть перенос бита, который равен 1, если два или больше входных бита равны 1, и равен 0 в противном случае.

Таблица истинности полного сумматора:



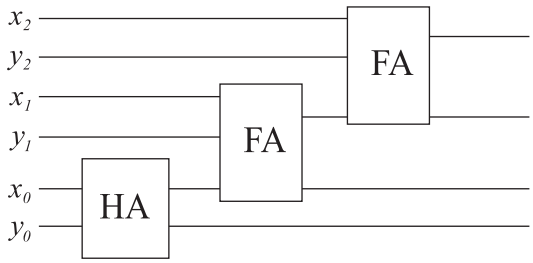
x	y	c	x	y	c	x	y	c	перенос	$x \oplus y \oplus c$	число
0	0	0	0	0	0	0	0	0	0	0	00
0	1	0	0	1	0	0	0	1	0	1	01
1	0	0	0	1	0	0	0	1	0	1	01
1	1	0	1	0	0	1	0	0	1	0	10
0	0	1	0	0	1	0	0	1	0	1	01
0	1	1	0	1	1	0	1	0	1	0	10
1	0	1	0	1	1	0	1	0	1	0	10
1	1	1	1	0	1	1	0	1	1	1	11

Каскад полных сумматоров, позволяющий построить цепь для сложения двух 2-х битовых целых $(x_1x_0) + (y_1y_0)$



x_1	x_0	y_1	y_0	x_1	x_0	y_1	y_0	c	x_0	y_0	$x+y$	10-тичное число
0	0	0	0	0	0	0	0	0	0	0	000	0
0	1	0	0	0	0	0	1	0	0	1	001	1
1	0	0	0	1	0	0	0	0	1	0	010	2
1	1	0	0	1	0	0	1	0	1	1	011	3
0	0	0	1	0	0	0	1	0	0	1	001	1
0	1	0	1	0	1	0	0	0	1	0	010	2
1	0	0	1	1	0	0	1	0	1	1	011	3
1	1	0	1	1	1	0	0	1	0	0	100	4
0	0	1	0	0	0	1	0	0	1	0	010	2
0	1	1	0	0	0	1	1	0	1	1	011	3
1	0	1	0	1	0	1	0	1	0	0	100	4
1	1	1	0	1	0	1	1	1	0	1	101	5
0	0	1	1	0	0	1	1	0	1	1	011	3
0	1	1	1	0	1	1	0	1	0	0	100	4
1	0	1	1	1	0	1	1	1	0	1	101	5
1	1	1	1	1	1	1	0	1	1	0	110	6

Каскад полных сумматоров позволяет построить цепь для сложения двух n-битовых целых. Пример для $n = 3$ приведен на рисунке ниже.



Здесь два трехбитовых целых числа представлены в виде $x = x_2x_1x_0$ и $y = y_2y_1y_0$. Аналогично может быть построена цепь вычисления произвольной функции.

В классическом компьютере NOT и NAND-гейты осуществляются транзисторами, например, как показано на рис. 6.1.

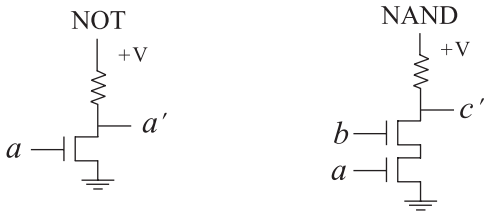


рис. 6.1.

Минимальная свободная энергия, которая расходуется на то чтобы работал идеальный компьютер зависит от набора и числа примитивных элементов. Например, на гейте NAND, выходящая линия c' принимает одно из двух значений и при этом энтропия изменяется на $\ln 2$ единицы. Теоретический минимум количества теплоты, которое рассеивается в пространство на одном элементарном шаге составляет $kT\ln 2$ (здесь k—постоянная Больцмана, T—абсолютная температура). Фактически в реальных вычислительных устройствах происходит диссипация энергии порядка $\sim 10^{10}\,kT$. Физически это связано с тем, что для изменения потенциала (напряжения) проводника он сначала заземляется через сопротивление, а затем через сопротивление заряжается.

Однако было установлено[1], что предел $kT\ln 2$ не является абсолютным, так как нет необходимости использовать в вычислительном устройстве необратимые гейты. Оказалось, что все операции, требующиеся для проведения вычислений могут быть проведены обратимым образом, а следовательно без диссипации энергии (в соответствии с законами термодинамики).

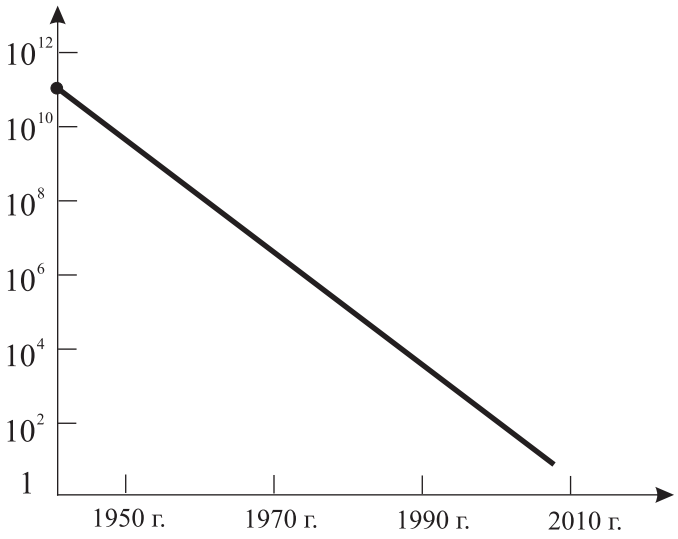


рис.6.2.

Классические компьютеры построены на электрических цепях, содержащих миллионы транзисторов. На рисунке 6.2. приведены результаты[2] оценки изменения числа примесей в основаниях биполярных транзисторов, требующихся для формирования логических операций в зависимости от времени развития полупроводниковых технологий.

По сути график отражает число электронов, необходимых для хранения одного бита информации. В соответствии с этими данными ясно, что технология достигает субатомные расстояния и фактически переходит на построение компьютеров на атомном и молекулярном уровне, что приводит к необходимости учета и включения квантово-механических свойств вещества. Как будет показано ниже, на квантовых компьютерах программы выполняются посредством организации унитарной эволюции входных данных на квантовых объектах. А так как унитарные преобразования обратимы, то цепь логических гейтов должна основываться на обратимых операциях.

[1]R. Landaner. IBM. J.Res. Develop. 3, p.183 (1961)

[2]R.W. Keyes. IBM. J.Res. Develop. 32, 24 (1988)

6.3 Обратимые логические гейты

Условием обратимости детерменированных устройств является возможность восстановления входных и выходных данных друг из друга единственным образом. Если в дополнение к логической обратимости устройство может функционировать в обратном направлении по времени, тогда оно называется физически обратимым и второй закон термодинамики гарантирует, что оно не рассеивает теплоту.

При построении логических цепей с обратимыми логическими элементами можно использовать три обратимых гейта [3]: NOT, CNOT, CCNOT, определение которых имеет следующий вид:

а. **NOT-гейт**. Стандартный NOT-гейт, очевидно не теряет информации и является обратимым. Обращение достигается повторным действием NOT-гейта.

б. **CNOT-гейт** (CNOT ≡ CONTROLLED NOT ≡ контролируемое "не").

Данный гейт определяется следующей диаграммой и таблицей истинности

a	b	a'	b'
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

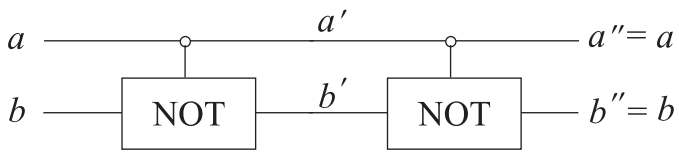
рис. 6.3.

Гейт CNOT имеет две входящие линии a и b (два входных бита) и две выходящие линии (две результирующих бита). Бит a' всегда тот же, что и a, а соответствующая линия называется контролирующей (или контрольной) линией. Если контролирующая линия активирована ($a = 1$), тогда выходной бит b' есть NOT от b. В противном случае ($a = 0$) $b' = b$:

$$b' = \begin{cases} NOT\, b, & \text{если } a = 1 \\ b, & \text{если } a = 0. \end{cases} \tag{6.14}$$

Используя определение гейта CNOT видно, что:

– действие данного гейта обращается простым повторением



a	b	a'	b'	a''	b''
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	1	1	0
1	1	1	0	1	1

рис. 6.4.

[3] T. Toffoli. Math. Syst. Theory 14, 13-23, 1981

– величина b' является симметричной функцией a и b и является гейтом XOR (исключающее или) a или b, но не оба. Данная операция является операцией суммирования a и b по модулю 2, что символически можно записать в виде

$$XOR : (x, y) \rightarrow (x,\, x \oplus y). \tag{6.15}$$

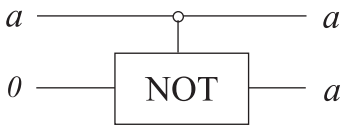
Соответственно символическое обозначение гейта CNOT допустимо писать в виде:

$$CNOT : (a, b) \rightarrow (a,\, a \oplus b) \equiv (a,\, XOR : (a, b)). \tag{6.16}$$

Подчеркнем также, что сам по себе гейт XOR не является обратимой логической операцией, так как, например, если результирующее значение гейта XOR равно 0, то нельзя определить из какого входного наборов битов $(a, b) = (0, 0)$ или $(a, b) = (1, 1)$ оно произошло.

Соответственно гейт CNOT сохраняет линию $a = a'$, что и устраняет неопределенность гейта XOR.

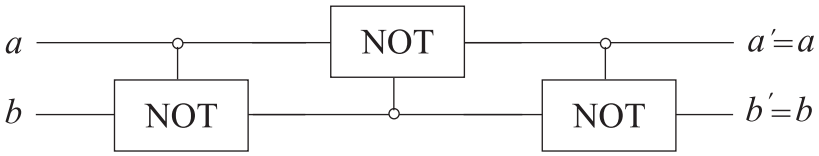
– гейт CNOT обеспечивает гейт FANOUT, так как при $b = 0$, a копируется на линию b'.

$FANOUT \equiv$ 

; или $FANOUT \equiv CNOT : (a, 0)$. (6.17)

Иногда функция копирования обозначается специальным символом COPY.

– гейт CNOT обеспечивает гейт EXCHANGE (или SWAP)

$EXCHANGE \equiv$ 

в. Гейт CCNOT ≡ (CONTROLLED CONTROLLED NOT) ≡ (контролируемое контролируемое "не"). Данный гейт определяется следующей диаграммой и таблицей истинности:

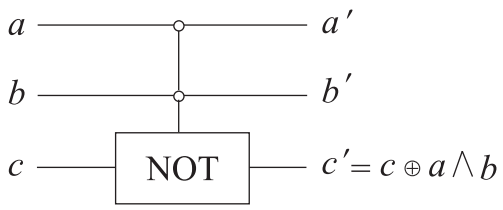


рис. 6.5.

a	b	c	a'	b'	c'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Гейт CCNOT – содержит три линии из которых две линии a и b являются контрольными и остаются неизменными на выходе, а выход третьей линии $c' = NOT\, c$, если обе контрольные линии активированы ($a = b = 1$), в противном случае $c' = c$.

Из определения гейта CCNOT следует, что

– если на входе третьей линии $c = 0$, то $c' = 1$ только при условии $a = 1$ и $b = 1$, в противном случае $a = 0$ или $b = 0$ или $a = b = 0$ на выходе третьей линии получим $c' = 0$. Таким образом в этом случае ($c = 0$) гейт определяет (функцию) гейт AND, что символически можно записать следующим образом:

$$CCNOT:\ (a, b, 0)\ \rightarrow\ (a, b,\ AND(a, b)). \tag{6.18}$$

Три комбинации входных битов (a, b), а именно $(0, 0)$, $(0, 1)$, $(1, 0)$, приводят к одному выходному биту логической функции $\mathrm{AND}(a, b) = 0$. Следовательно для устранения неоднозначности требуется помнить оба входных бита или иметь две контролируемые линии.

Сохранение этих битов на линиях a, b на выходе позволяет обратить действие гейта.

– повторное выполнение операции CCNOT обращает результат первой операции CCNOT

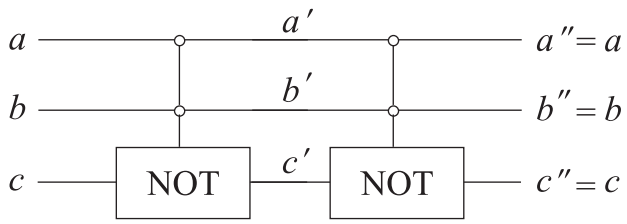


рис. 6.6.

a	b	c	a'	b'	c'	a''	b''	c''
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0
0	1	1	0	1	1	0	1	1
1	0	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	0
1	1	1	1	1	0	1	1	1

Из определения CCNOT-гейта (или Тоффоли-гейта) видно, что выход этого гейта может быть разложен в различные гейты

$$c \oplus a \wedge b = \begin{cases} a \wedge b, & \text{для } c = 0 \quad \text{(AND-гейт)} \\ a \oplus c, & \text{для } b = 1 \quad \text{(XOR-гейт)} \\ \bar{c}, & \text{для } a = b = 1 \quad \text{(NOT-гейт)} \\ a, & \text{для } b = 1,\ c = 0 \quad \text{(FANOUT-гейт)} \end{cases} \tag{6.19}$$

То есть данный гейт является универсальным, поскольку он выполняет AND, XOR, NOT и FANOUT, в зависимости от того, что имеется на входе.

Комбинируя обратимые логические элементы можно составить любую логическую схему и построить универсальный компьютер. Например, используя последовательность CCNOT и CNOT можно составить полусумматор (HA) рис. 6.7.

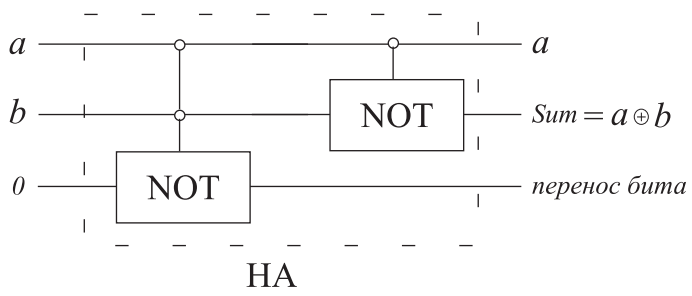


рис. 6.7.

Полный сумматор (FA), который использует перенос c (от предыдущего суммирования) и складывает его с двумя битами (линиями) a и b, а кроме того, содержит дополнительную d с равным нулю битом на вхаде можно построить на основе четырех гейтов (рис. 6.8.)

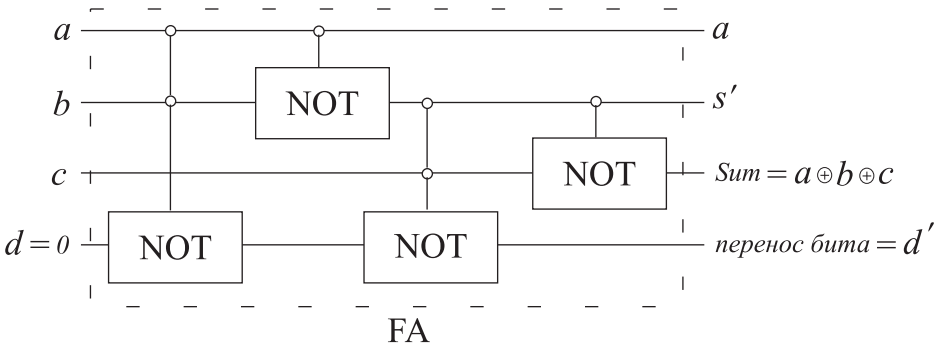


рис. 6.8.

Помимо полной суммы трех битов $a \oplus b \oplus c$ по модулю 2 и переноса бита в полном сумматоре (рис. 6.8.) присутствуют еще два сообщения на линиях a и b. При этом $a \rightarrow a$, $b \rightarrow s' = a \oplus b$– промежуточная сумма. Это обстоятельство является типичным для организации логики вычислений на обратимых гейтах. Как видно, на выходе получается не только то, что требовалось получить (Sum и перенос бита), но и определенное количество промежуточной информации, которую принято называть "мусором". Наличие мусора в таких цепях является проблемой, так как разрастание цепей до миллионов гейтов означает необходимость хранения огромного количества ненужной информации и неэффективное использование технологических элементов компьютера. Поэтому при организации процесса вычисления на обратимых элементах необходимо еще решать задачу борьбы с мусором. В конкретном случае, представленном на рис. 6.8., мусор может быть сведен в точности к тому, что имеется на выходе, если к блоку FA добавить дополнительно CNOT на две верхние линии.

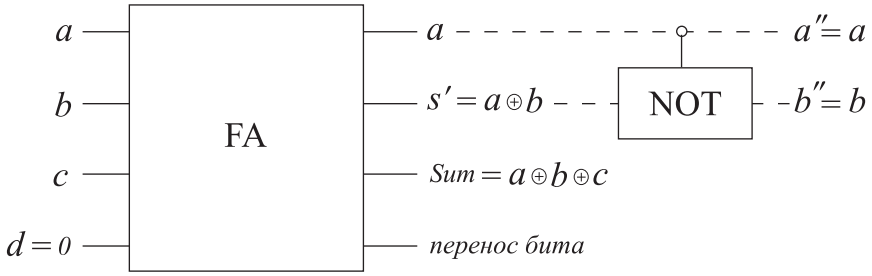


рис. 6.9.

Действительно, если $a = 0$, то $s' = a \oplus b = b$ и в соответствии с определением гейта CNOT (т.к. линия a не активизирована) $b'' = b$. Во втором возможном случае $a = 1$, $s' = 1 \oplus b$. При этом:

$$s' = \begin{cases} 1, & \text{если } b = 0 \\ 0, & \text{если } b = 1. \end{cases} \tag{6.20}$$

В силу того, что линия $a = 1$ (активизирована), результат действия гейта CNOT на линии $b \equiv s'$ будет равен:

$$b'' = \begin{cases} NOT\, s' = 0, & \text{если } b = 0 \\ NOT\, s' = 1, & \text{если } b = 1 \end{cases} = b. \tag{6.21}$$

Практически аналогично мусор может быть удален во всех иных схемах логических цепей. В общем случае схема, представленная на рис. 6.9. может быть упрощена, но для иллюстрации принципов это не существенно.

Таким образом, составляя различные цепи из комбинаций обратимых гейтов, можно построить общий логический блок, который преобразует n-битов обратимым образом. Если сама решаемая задача обратима, тогда может не быть мусорной информации, но в общем случае необходимы дополнительные линии (или биты), которые требуются для обращения выполняемой операции. В этом смысле мусор содержит информацию, необходимую для обращения процесса вычисления.

Ранее (формула (6.17) и обратимость гейта CNOT) было показано, что гейт FANOUT является обратимым. Очевидно, что гейт FANOUT не разрушает информацию, поэтому возможна его обратимость.

Теперь рассмотрим операцию ERASE, которая, как кажется, требуется для периодической "очистки" (обнуления) памяти компьютера. Интересно, что один тип стирания может быть осуществлен обратимым образом. Действительно, если есть продублированная копия некоторого бита, то можно стереть добавочную копию (или копию) путем операции обратной FANOUT-гейту.

Проблема возникает, когда стирается имеющаяся последняя копия, в этом случае говорят о так называемом примитивном стирании. К счастью гейт примитивный ERASE не является абсолютно необходимым в вычислениях.

Действительно, вычисление произвольной функции $f(a)$ может быть воспроизведено взаимно однозначным соответствием с ее агрументом в результате сохранения копии входных данных a:

$$f\ :\ a\ \rightarrow\ (a,\ f(a)) \tag{6.22}$$

Процедура вычисления приводит к прямой проблеме из-за отсутствия примитивного ERASE. Чем больше гейтов используется, тем больше мусорных битов накапливается, так как в каждом гейте сохраняются входные биты для обеспечения обратимости. Таким образом, компьютер, построенный из логически обратимых гейтов ведет себя следующим образом:

$$f\ :\ a\ \rightarrow\ (a,\ j(a),\ f(a)), \tag{6.23}$$

где $j(a)$– большое число дополнительных мусорных битов.

Беннет решил эту проблему, путем стирания мусорных битов обратимым образом на промежуточных шагах вычисления. Идея решения проблемы состоит в использовании следующей процедуры:

1. На первом шаге вычисляется f и при этом получаются мусорные биты $j(a)$ и искомый результат (6.23).

2. На втором шаге применяется FANOUT-гейт для дублирования выходного результата $f(a)$ в $f_c(a)$

$$FANOUT : (a, j(a), f(a)) \rightarrow (a, j(a), f(a), f_c(a)). \tag{6.24}$$

3. На третьем этапе выполняется операция обратная вычислению функции f

$$f^{-1} : (a, j(a), f(a), f(a)) \rightarrow (a, f_c(a)). \tag{6.25}$$

Обратная операция удаляет как мусорные биты, так и биты первоначально вычисленного выходного результата $f(a)$, не трогая при этом копию выходного результата $f_c(a)$.

Таким образом, в памяти машины остается только набор начальных данных a и копия набора битов вычисленной функции $f(a)$. При этом использования примитивного ERASE не потребовалось.