Организация ЭВМ и вычислительных систем

Практические занятия №3–7. Логические основы вычислительной техники

Макаров П. А.

Содержание

Цель 1

1 Краткая теория 2
- 1.1 Основы булевой алгебры . . . 2
- 1.2 Логические функции . . . 4
- 1.3 Связь булевой алгебры с цифровой электроникой . . . 5
  - 1.3.1 Временны́е характеристики цифровых схем . . . 7
  - 1.3.2 Третье состояние и шинная архитектура . . . 8
  - 1.3.3 Логические и побитовые операции . . . 8
  - 1.3.4 Безопасные типы и неопределённое поведение при сдвигах . 9
  - 1.3.5 Регистр флагов `RFLAGS` и инструкции `test`/`cmp` . . . 10
  - 1.3.6 Шаблоны побитовых операций в ассемблере `nasm` для x86-64 10
  - 1.3.7 Ключевое слово `volatile` и отображение регистров в память 11

2 Практические задания и контрольные вопросы 11

Источники 14

Цель

- Освоение математического аппарата булевой алгебры и методов синтеза комбинационных логических цепей (таблицы истинности, карты Карно, функциональная полнота).
- Изучение аппаратной реализации логических операций: от базовых вентилей и триггеров до регистров, SRAM и арифметико-логических узлов (сумматоры), а также принципов их взаимодействия в вычислительной системе.

- Приобретение практических навыков проектирования цифровых схем в САПР Logisim, программного моделирования логических устройств на языке C.

- Исследование соответствия высокоуровневых побитовых операций машинным инструкциям архитектуры x86.

1 Краткая теория

1.1 Основы булевой алгебры

Раздел математической логики, изучающий логические операции над высказываниями[1] называется *алгеброй логики* (или *алгеброй высказываний*).

Пусть x, y, z и т. д. — есть некие *элементарные логические высказывания*, о которых достоверно можно утверждать только истинны (`True`) они или ложны (`False`). В этом случае алгебра соответствующих высказываний также оказывается *двоичной* (или *бинарной*). Именно такой вариант алгебры логики называют *булевой алгеброй* (по имени Джорджа Буля — первооткрывателя этой науки).

Часто говорят, что значению `True` соответствует *логическая единица* «1», а `False` — *логический ноль* «0». Здесь важно подчеркнуть, что в разных языках программирования поддержка булевой алгебры реализована по-разному. Так, в ранних стандартах языка C не было предусмотрено специального булева (boolean) типа[2], поскольку предполагалось, что это избыточно, т. к. любое не равное нулю значение произвольного выражения всегда считается истиной (`True`), в то время как нулевое значение — это ложь (`False`). В связи с этим интересно проанализировать фрагмент исходного кода на языке C, представленный в Листинге 1.

```c
int x = 2;
if (x)
  printf("A\n");
if (x = 0)        // Второй вариант с присваиванием: if (x = 1)
  printf("B\n");
if (x == 0)
  printf("C\n");
printf("x = %d\n", x);
```

Листинг 1: Интерпретация `True` и `False` в C

Формально, булева алгебра определятся с помощью следующей алгебраической структуры: $\{B, \neg, \wedge, \vee, 0, 1\}$, где B — непустое множество, над элементами которого $(\forall x, y \in B)$ определены три операции:

[1] Высказываниями в математической логике называются любые предложения, выражающие то или иное суждение.

[2] Хотя, конечно, его всегда можно либо сконструировать самостоятельно, либо использовать дополнительные библиотеки, представляющие такую возможность. Кроме того, начиная со стандарта C99 в языке появился встроенный тип `_Bool`, а заголовочный файл `<stdbool.h>` макросами `bool`, `true`, `false` обеспечивает привычный синтаксис.

- отрицание (унарная операция): $\neg x$ (или $\overline{x}$);
- конъюнкция (бинарная операция): $x \wedge y$ (или $x \cdot y$);
- дизъюнкция (бинарная операция): $x \vee y$ (или $x + y$);

а 0 и 1 — константы, означающие `False` и `True`, соответственно.

Основы булевой алгебры кратко изложены в таблице 1. Первые девять строк этой таблицы представляют собой определения и аксиомы булевой алгебры, в то время как остальные строки демонстрируют некоторые свойства логических операций: идемпотентность (10), коммутативность (11), ассоциативность (12), закон поглощения (13), правила де Моргана (17) и некоторые другие.

Таблица 1: Основы булевой алгебры

(1)	$x = 0$, если $x \neq 1$		$x = 1$, если $x \neq 0$
(2)	$x = 0, \overline{x} = 1$		$x = 1, \overline{x} = 0$
(3)		$\overline{\overline{x}} = x$	
(4)	$1 \cdot 0 = 0 \cdot 1 = 0$		$1 + 0 = 0 + 1 = 1$
(5)	$1 \cdot 1 = 1$		$0 + 0 = 0$
(6)	$0 \cdot 0 = 0$		$1 + 1 = 1$
(7)	$x \cdot 1 = x$		$x + 0 = x$
(8)	$x \cdot 0 = 0$		$x + 1 = 1$
(9)	$x \cdot \overline{x} = 0$		$x + \overline{x} = 1$
(10)	$x \cdot x = x$		$x + x = x$
(11)	$x \cdot y = y \cdot x$		$x + y = y + x$
(12)	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		$(x + y) + z = x + (y + z)$
(13)	$x + x \cdot y = x$		$x \cdot (x + y) = x$
(14)		$(x + \overline{y}) \cdot y = x \cdot y$	
(15)	$x \cdot y + x \cdot \overline{y} = x$		$(x + y)(x + \overline{y}) = x$
(16)	$(x + y) \cdot (x + z) = x + y \cdot z$		$x \cdot y + x \cdot z = x \cdot (y + z)$
(17)	$\overline{x \cdot y \cdot z} = \overline{x} + \overline{y} + \overline{z}$		$\overline{x + y + z} = \overline{x} \cdot \overline{y} \cdot \overline{z}$

Несмотря на упомянутую выше неоднозначность трактовки понятия `True` в языке C (см. Листинг 1), аппарат булевой алгебры реализован в нём полностью с помощью следующих логических операций: `!` (отрицание), `&&` (конъюнкция) и `||` (дизъюнкция). Демонстрационный пример на эту тему представлен в Листинге 2.

```c
#include <stdio.h>

int f(int, int, int);

int main() {
  int i;
  printf("| x | y | z | f |\n");
  printf("|---+---+---+---|\n");
  for (i = 0; i < 8; i++)
    printf("| %d | %d | %d | %d |\n", i&1, i&2, i&4, f(i&1, i&2, i&4));
  return 0;
}
```

```c
int f(int x, int y, int z) {
  return x || !y && z;  // Важно знать приоритеты логических операций!
}
```

Листинг 2: Пример использования булевой алгебры (и логических функций) в C

1.2 Логические функции

С помощью логических операций из исходных высказываний $(x, y, z, \dots)$ можно построить более сложные логические предложения, которые называют *логическими функциями* $f(x, y, z, \dots)$. Логические функции на практике часто описывают с помощью так называемых *таблиц истинности*. Например, вместо того, чтобы определять операцию «логическое отрицание» чисто аксиоматически (см. строки (1) – (3) в таблице 1), можно построить функцию $f(x) = \overline{x}$, определив её с помощью таблицы 2.

Таблица 2: Таблица истинности для функции, реализующей «отрицание»

x	$f(x) = \overline{x}$
0	1
1	0

Аналогично можно построить логические функции двух (и более) логических переменных. Такая точка зрения позволяет трактовать конъюнкцию и дизъюнкцию как пару возможных примеров различных логических функций двух переменных, наравне с которыми можно рассматривать и другие (см. таблицу 3).

Таблица 3: Таблица истинности некоторых логических функций двух переменных

x	y	Конъюнкция $x \wedge y$ ($\cdot$, &&)	Дизъюнкция $x \vee y$ ($+$, ||)	Исключение $x \oplus y$	Импликация $x \to y$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	1

Принципиально то, что одну и ту же логическую функцию можно реализовать по-разному. Так, опираясь на формальное определение булевой алгебры, данное выше (т. е. считая элементарными бинарными операциями конъюнкцию и дизъюнкцию), можно представить функцию, реализующую исключение следующим

образом:

$$f(x,y) \equiv x \oplus y = (x \vee y) \wedge \overline{x \wedge y}. \tag{1}$$

Вместе с тем, не противоречива и иная точка зрения, согласно которой элементарными бинарными операциями можно считать дизъюнкцию и исключение. Тогда функцию, выполняющую конъюнкцию можно выразить так:

$$f(x,y) \equiv x \wedge y = \overline{x \vee y} \oplus \overline{x \oplus y}. \tag{2}$$

Приведённые соображения приводят к формированию понятия *функционально полной системы логических элементов*, как такого набора элементарных логических операций (называемых элементами), с помощью которого можно реализовать любую сколь угодно сложную логическую функцию.

Очевидно, что набор из трёх элементов, реализующих операции отрицание («НЕ», `NOT`), конъюнкция («И», `AND`) и дизъюнкция («ИЛИ», `OR`) является функционально полным. Однако для построения функционально полного набора можно использовать и другие элементы, реализующие логические функции, отличные от простейших. В частности, можно построить полный набор из всего одного элемента, например «ИЛИ-НЕ»(`NOR`) или «И-НЕ» (`NAND`).

1.3 Связь булевой алгебры с цифровой электроникой

Логические элементы также часто называют вентилями или гейтами (от англ. logic gates). Именно эти устройства лежат в основе функционирования подавляющего большинства современных компьютеров[3]. Так, в принципе можно считать, что на самом низком уровне абстракции компьютер — это устройство, способное выполнять произвольные логические функции над входными двоичными данными с помощью тех или иных вентилей, некоторые примеры которых приведены на рис. 1.

Более того, сами двоичные числа, над разрядами которых процессором компьютера выполняются заданные логические функции, хранятся в ячейках построенных в свою очередь также из логических элементов. Так, *триггерами* (см. пример на рис. 2) называют логические элементы с двумя устойчивыми логическими состояниями («0» и «1»), которые можно использовать как элементы памяти для хранения одного бита информации. Именно на триггерах реализуется так называемая статическая память с произвольным доступом (SRAM), используемая, в частности, в качестве кэш-памяти процессоров. Триггеры, объединённые в группы по 8, 16, 32, 64, и т. д. составляют регистры. Таким образом, регистры могут хранить двоичное число определённой разрядности.

Комплексным примером, иллюстрирующим всё сказанное в этом разделе, является рис. 3, на котором показано как можно осуществить сложение двух восьмиразрядных двоичных чисел с помощью четырёх устройств, принципы действия которых описываются булевой алгеброй.

[3]Здесь мы сознательно ограничиваемся рассмотрением только классических цифровых электронных вычислительных устройств, работающих с двоичным кодом.

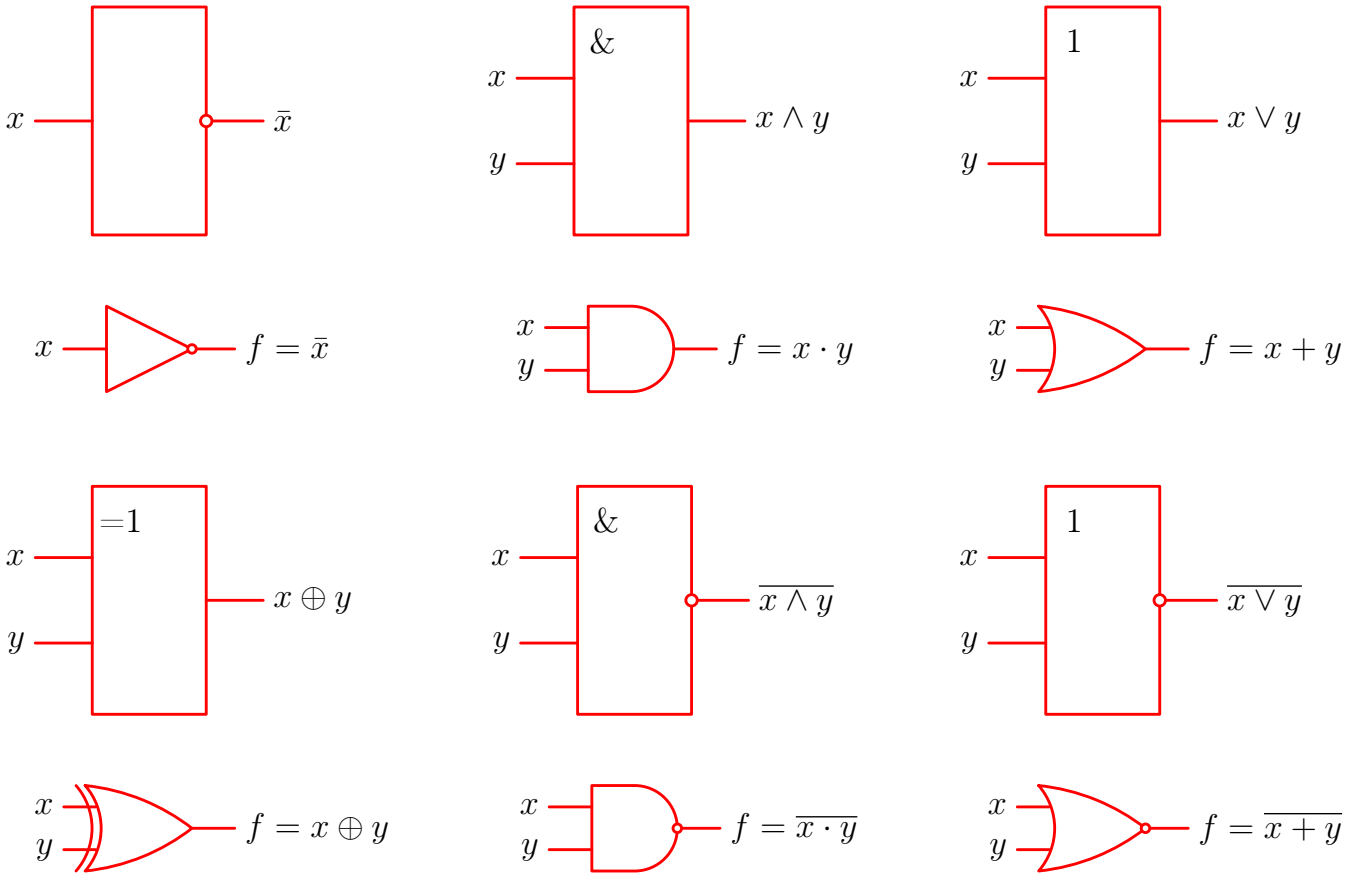


Рис. 1: Схемные обозначения некоторых вентилей

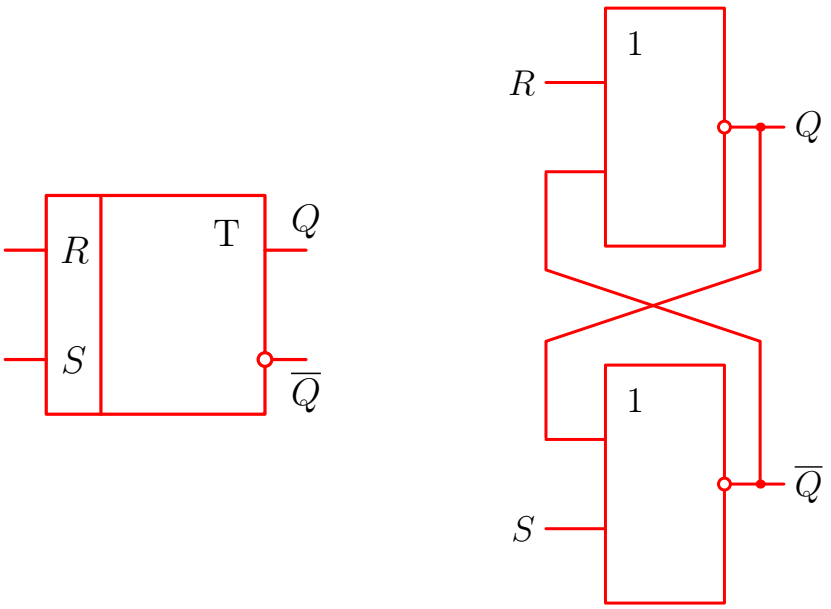


Рис. 2: Схемное обозначение RS-триггера и его реализация в базисе ИЛИ-НЕ

В верхней части рис. 3 изображены два восьмиразрядных регистра `X` и `Y`, хранящие операнды. В нижней части схемы размещается восьмибитный регистр `Z`, в который будет записан результат следующей инструкции.

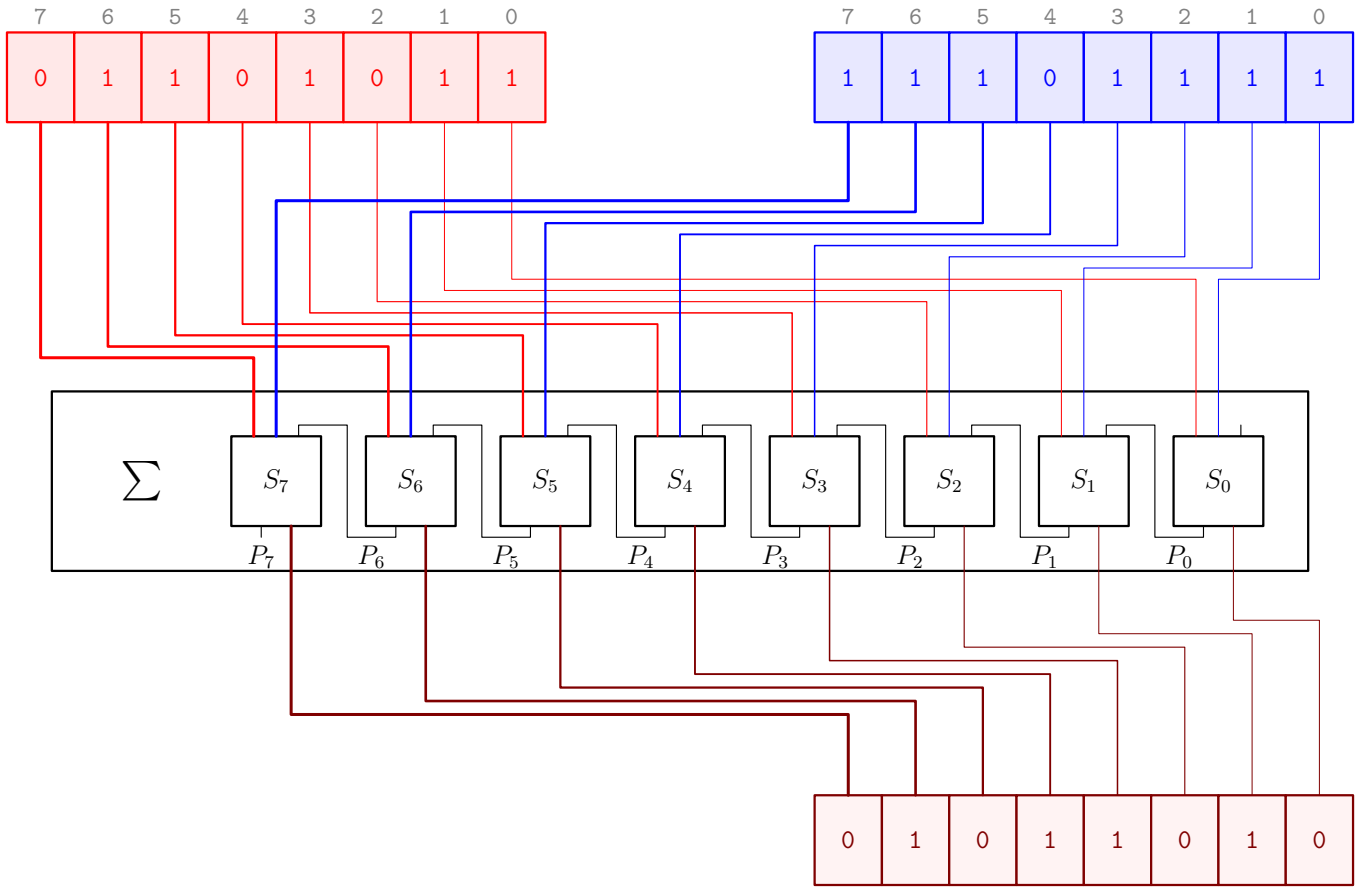


Рис. 3: Суммирование двух восьмиразрядных чисел

```
1    ADD Z, X, Y     ; Z = X + Y
```

Каждый регистр `X`, `Y` и `Z` в этой схеме построен из тех или иных восьми триггеров — *последовательностных* логических устройств с памятью. Устройство Σ реализующее инструкцию сложения `ADD`, показано в центре рис. 3 как цепь из восьми последовательно соединённых полных сумматоров — *комбинационных* логических цепей[4]. Принцип действия полного сумматора представлен в таблице истинности 4.

1.3.1 Временны́е характеристики цифровых схем

При переходе от абстрактной булевой алгебры к физической реализации логических функций на непосредственных устройствах цифровой электроники необходимо учитывать *временны́е задержки*. Так, любое логическое устройство характеризуется следующими параметрами:

[4]Несложно понять, что такая схема сумматора Σ обладает линейной задержкой распространения переноса $O(n)$, поэтому в современных процессорах заменяется схемами с ускоренным переносом. В качестве таковых используется, например, CLA (Carry-Lookahead Adder) — сумматор с ускоренным (предвосхищающим) переносом. В отличие от классического последовательного сумматора (Ripple-Carry), где сигнал переноса распространяется от младшего разряда к старшему с задержкой $O(n)$, CLA вычисляет все переносы параллельно с помощью дополнительных логических цепей генерации G_i и распространения P_i. Это сокращает задержку сложения до $O(\log n)$ (или константы при каскадировании блоков), что является стандартом для арифметико-логических устройств (АЛУ) современных процессоров.

Таблица 4: Принцип функционирования полного двоичного сумматора

x_i	y_i	P_{i-1}	S_i	P_i
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

- t_{pd} (propagation delay) — время задержки распространения сигнала от входа до выхода вентиля;
- t_{su} (setup time) — минимальное время, в течение которого входной сигнал должен оставаться стабильным *до* активного фронта тактового импульса;
- t_{h} (hold time) — минимальное время стабильности входного сигнала *после* активного фронта импульса синхронизации.

Нарушение условий t_{su} или t_{h} приводит к *состоянию гонки* (race condition) и неопределённому поведению триггеров. В среде Logisim данные эффекты моделируются автоматически при включении режима тактирования (`Tick Frequency`), что позволяет наблюдать разницу между мгновенным срабатыванием комбинационных цепей и синхронным обновлением последовательностных устройств.

1.3.2 Третье состояние и шинная архитектура

Помимо логических «0» и «1», в цифровой электронике используется *третье состояние* (High-Impedance, обозначается `Z`). В этом состоянии выход вентиля электрически отключается от схемы, что позволяет нескольким источникам данных совместно использовать одну шину без коротких замыканий. Управление осуществляется через дополнительный вход разрешения (`EN` или `OE`).

1.3.3 Логические и побитовые операции

Очень важно отличать логические операции от схожих с ними *побитовых операций*. Последние в языке C представлены следующими операциями: `~` (инверсия), `&` (`AND` — поразрядное И), `|` (`OR` — поразрядное ИЛИ), `^` (`XOR` — исключающее ИЛИ), `>>` (`SHR` — побитовый сдвиг вправо[5]), `<<` (`SHL` — побитовый сдвиг влево). В

[5]На большинстве платформ (включая x86/x86-64) выполняется как арифметический сдвиг с сохранением знакового бита, однако согласно стандарту C результат для отрицательных чисел является implementation-defined. Обратите в связи с этим на результаты работы строк 12 и 13 в Листинге 3. Как вы думаете, почему это реализовано именно так?

связи с этим полезно проанализировать результат работы программы, исходный код которой представлен в Листинге 3.

```c
#include <stdio.h>

void char2bin(char);

int main() {
  char x = 0xA5;
  char y = 0x0F;
  printf("   x   = "); char2bin(x);
  printf("  ~x   = "); char2bin(~x);
  printf("   y   = "); char2bin(y);
  printf("x & y  = "); char2bin(x&y);
  printf("x | y  = "); char2bin(x|y);
  printf("x ^ y  = "); char2bin(x^y);
  printf("x >> 1 = "); char2bin(x>>1);
  printf("y >> 1 = "); char2bin(y>>1);
  printf("x << 1 = "); char2bin(x<<1);
  printf("y << 1 = "); char2bin(y<<1);
  return 0;
}

void char2bin(char x) {
  int i = 128;
  while (i > 0) {
    printf("%d", (x & i) ? 1 : 0);
    i = i >> 1;
  }
  printf("\n");
  return;
}
```

Листинг 3: Пример использования побитовых операций в C

1.3.4 Безопасные типы и неопределённое поведение при сдвигах

Для точного моделирования аппаратных регистров в языке C настоятельно рекомендуется использовать заголовочный файл `<stdint.h>`, предоставляющий типы фиксированной ширины (`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`). Использование базовых типов `int` или `long` переносимо только на уровне стандарта, но фактическая ширина зависит от платформы.

Важно: Сдвиги влево/вправо на значение, большее или равное ширине типа, а также сдвиг отрицательных знаковых чисел влево, согласно стандарту C, приводят к *неопределённому поведению* (UB — Undefined Behavior). Компилятор не обязан диагностировать такие случаи, но код может давать непредсказуемые результаты или оптимизироваться некорректно.

1.3.5 Регистр флагов RFLAGS и инструкции test/cmp

Арифметические и логические операции в архитектуре x86 не только изменяют операнды, но и обновляют биты регистра флагов (RFLAGS). Наиболее значимыми флагами для системного программирования являются следующие:

- CF (Carry Flag) — флаг переноса/заёма;
- ZF (Zero Flag) — устанавливается в 1, если результат операции равен нулю;
- SF (Sign Flag) — копирует старший (знаковый) бит результата;
- OF (Overflow Flag) — сигнализирует о переполнении знакового разряда;
- PF (Parity Flag) — чётность младшего байта результата.

Инструкции test и cmp выполняют, соответственно, логическое AND и вычитание (SUB), **но не сохраняют результат в операндах**, а только обновляют флаги. Это делает их идеальным инструментом для организации условных переходов (jz, jnz, jc, jo и т. д.) без изменения исходных данных.

1.3.6 Шаблоны побитовых операций в ассемблере nasm для x86-64

При разработке драйверов и низкоуровневых системных функций широко используются паттерны для работы битами, показанные в Листинге 4 (здесь n — номер бита, а нумерация, как и всегда, начинается с 0).

```
; Установка бита n (OR)
or  rax, (1 << n)

; Сброс бита n (AND с маской)
and rax, ~(1 << n)

; Инверсия бита n (XOR)
xor rax, (1 << n)

; Проверка бита n без изменения регистра (TEST)
test rax, (1 << n)
jnz  bit_is_set    ; ZF=0, бит установлен
jz   bit_is_clear  ; ZF=1, бит сброшен
```

Листинг 4: Пример использования побитовых операций в nasm для x86-64

Обратите внимание: в ассемблере nasm операторы << и ~ вычисляются на этапе ассемблирования (assembly-time). В отладчике или машинном коде им соответствуют непосредственные значения (immediates) соответствующих битовых масок.

1.3.7 Ключевое слово `volatile` и отображение регистров в память

При работе с периферией, регистры которой отображаются в адресное пространство процессора (Memory-Mapped I/O), компилятор может некорректно оптимизировать обращения к ним. Например, цикл опроса статуса устройства может быть удалён или выполнен один раз, если компилятор сочтёт, что значение по указателю не меняется. Квалификатор `volatile` запрещает такие оптимизации, гарантируя, что каждое обращение к переменной будет выполнено физически:

```c
#include <stdint.h>
volatile uint32_t * const PORT_STATUS = (uint32_t*)0xF0000000;

void wait_ready(void) {
  // Цикл не будет оптимизирован компилятором
  while (!(*PORT_STATUS & (1U << 3))) ;
}
```

Листинг 5: Пример применения побитовых операций при работе с периферией

Примечание: данный пример показан для bare-metal/embedded сред. В ОС общего назначения доступ реализуется через `mmap()/ioremap()`.

2 Практические задания и контрольные вопросы

1. Что такое алгебра логики и булева алгебра? Есть ли разница между этими терминами? Как ответы на эти вопросы соотносятся с определениями в строке (1) таблицы 1 и материалом, приведённым в разделе 1.1? Как физическая реализация цифровых логических устройств (включающая третье высокоимпедансное состояние, см. раздел 1.3.2) расширяет строгие рамки математической модели из строки (1) таблицы 1?

2. Проанализируйте оба варианта исходных кодов, представленных в Листинге 1. Что будет напечатано на экране в обоих случаях? Почему?

3. Что называют логическим вентилем? Какие они бывают? Что такое функционально полный набор и базис? Докажите функциональную полноту двух наборов элементов: `NOR` и `NAND`.

4. Сколько всего возможно двухвходовых логических вентилей? Что такое штрих Шеффера $\uparrow$ (иногда также обозначается как $|$, но не следует путать этот символ с операцией `|` поразрядного ИЛИ, описанной в разделе 1.3.3) и стрелка Пирса $\downarrow$? Расширьте таблицу истинности 3, добавив в неё информацию о всех возможных двухвходовых гейтах.

5. Постройте таблицы истинности для функций:

   - $f(x, y, z) = x \cdot y + \overline{z}$;

- $f(x, y, z) = (x + y) \cdot (\overline{x} + z)$;
- $f(x, y, z) = x \oplus y \oplus z$.

6. Исследуйте работу программы, исходный код которой представлен в Листинге 2. Каковы приоритеты логических операций? Как можно изменять порядок их вычислений?

7. Какие схемы цифровой электроники называют комбинационными и последовательностными? Как эти виды схем соотносятся с соединениями логических вентилей?

8. Каковы основные этапы синтеза комбинационных логических цепей?

9. Минимизируйте следующие функции с помощью диаграмм Вейча (карт Карно):

   - $f(x_1, x_2) = \overline{x_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2} + \overline{x}_1 \cdot \overline{x}_2$;
   - $f(x_1, x_2, x_3) = \overline{(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + x_3)} + \overline{x}_1 x_2 x_3 + \overline{x}_2 x_3$;
   - $f(x_1, x_2, x_3, x_4) = \overline{x_1 + \overline{x}_1 x_3 \overline{x}_4 + \overline{x}_2 \overline{x}_3 \overline{x}_4} + x_1 x_2 x_3 x_4 + x_1 \overline{x}_2 x_3 \overline{x}_4 + x_1 x_2 \overline{x}_3 x_4$.

10. Составьте таблицу истинности RS-триггера. Какие ещё бывают триггеры?

11. Что такое Logisim? Какова история этого проекта и в каком состоянии он находится сейчас?

12. Синтезируйте простейшую пороговую ячейку (модель искусственного нейрона в элементарной реализации) — трёхвходовую логическую цепь, сигнал на выходе которой будет равен единице только тогда, когда на её входах присутствует не менее двух единичных сигналов. Сколько решений имеет эта задача?

13. Спроектируйте полусумматор — логическое устройство, которое вырабатывает сигналы суммы S и переноса P при сложении двух двоичных одноразрядных чисел.

14. Разработайте сумматор — логическую схему, выполняющую суммирование двух двоичных одноразрядных чисел x_i и y_i с учётом сигнала переноса из предыдущего разряда P_{i-1} (см. таблицу 4).

15. Реализуйте на языке C логику работы двух предыдущих устройств. Придумайте для этого необходимые структуры данных и напишите работающие с ними функции `halfadder` и `fulladder`. Продемонстрируйте результаты работы с помощью соответствующих таблиц. Решите задачу различными способами: ограничиваясь лишь средствами ANSI C и пользуясь возможностями стандартов C17/C23 и библиотеки `<stdint.h>`.

16. В чём именно состоят недостатки сумматора Σ, показанного в центре рис. 3? *Подсказка:* почему различные соединения на этом рисунке показаны линиями *разной толщины?* Как этот недостаток преодолевают в современной цифровой электронике? Также в контексте данной задачи ещё раз вернитесь к содержимому раздела 1.3.2 и модернизируйте схему передачи данных между регистрами `X`, `Y`, `Z` и устройством Σ, заменив выделенные проводные соединения на общую шину с управлением через третье состояние.

17. Как в контексте потенциального использования прямого и дополнительного кодов для операндов `X` и `Y` можно интерпретировать результаты суммирования, показанные на рис. 3? Опираясь на материал раздела 1.3.5, предложите способы улучшения данной схемы направленные на диагностику и обнаружение возможных ошибок.

18. Постройте логическую цепь, вычисляющую сумму двух одноразрядных десятичных чисел. Реализуйте два варианта представления результата: в двоичной системе счисления и в двоично-десятичном взвешенном коде 8-4-2-1. Предусмотрите дополнительный выход устройства E, используемый для сигнализации ошибки входной комбинации (пусть $E = 1$ когда хотя бы одно из чисел на входе выходит за рамки требуемого диапазона). Какие выводы вы можете сделать по результатам решения этой задачи?

19. Сконструируйте логическое устройство, выявляющее совпадение кодов. Пусть на вход цепи подаются два произвольных восьмиразрядных двоичных кода X и Y, а сигнал на выходе Z равен единице только в случае $X = Y$.

20. Проанализируйте результат работы программы, исходный код которой представлен в Листинге 3.

21. Ознакомьтесь с основными логическими побитовыми инструкциями, поддерживаемыми центральными процессорами семейства x86. В частности, найдите в документации принцип действия и соответствующие машинные коды для тех инструкций, ассемблерные мнемоники которых перечислены здесь: `and`, `or`, `xor`, `not`, `rcl`, `rcr`, `rol`, `ror`, `shl/sal`, `shr`, `sar`. Как вся эта информация соотносится с тем, что было представлено в Листингах 3 и 4?

22. **Анализ вывода компилятора.** Напишите простую функцию на C с использованием побитовых операций: маскирование, установка битов, сдвиги (можно взять за основу весь Листинг 3). Скомпилируйте написанный вами исходный код `source.c` в ассемблерный текст `result.asm` двумя способами.

    - Стандартно:

      ```
      $ gcc -S -masm=intel source.c -o result1.asm
      ```

    - и с оптимизацией:

      ```
      $ gcc -O2 -S -masm=intel source.c -o result2.asm
      ```

Сравните полученные результаты. Найдите в полученных файлах соответствующие инструкции `and`, `or`, `shl`, `test` и проанализируйте, как компилятор распределил их по регистрам и сгенерировал константы-маски.

23. Что происходит в Листинге 5? Объясните назначение квалификатора `volatile`, принцип работы битовой маски `(1U << 3)` и причину использования пустого оператора (одиночной `;`) в качестве тела цикла активного опроса (polling).

24. Известны ли вам примеры цифровых компьютеров, построенных на небинарной логике? Каковы их преимущества и недостатки?

25. Что такое квантовый компьютер? Являются ли кубиты (qubit) элементами, реализующими небинарную логику? Что вы знаете о кудитах (qudit)?

Источники

1. Patt Y.N., Patel S.J. Introduction to Computing Systems: From Bits & Gates to C/C++ & Beyond.
2. Брайант Р.Э., О'Халларон Д.Р. Компьютерные системы: архитектура и программирование (Computer Systems: A Programmer's Perspective).
3. В.С. Гутников. Интегральная электроника в измерительных приборах. — Л.: Энергия, 1974, 144 с.
4. Д.О. Квочкин, П.А. Макаров, В.А. Устюгов. Микросхемотехника. — Сыктывкар.: Изд-во СГУ им. Питирима Сорокина, Ч.1 – 2016, 64 с; Ч.2 – 2017, 67 с.
5. А.В. Карпов, А.С. Калабанов, Р.А. Ишмуратов. Основы цифровой электроники. — Казань.: Изд-во Казан. ун-та, 2019, 75 с.
6. Logisim Docs `https://www.cburch.com/logisim/docs.html`.
7. `https://logisim.app`.
8. Logisim-evolution.
9. Керниган Б., Ритчи Д. Язык программирования С.
10. О. Большаков. Выбираем стандарт языка Си в 2025 году.
11. Hyde R. The Art of Assembly Language.
12. Марек Р. Ассемблер на примерах. Базовый курс.
13. Tech Help Manual. 8088/286/386/486/Pentium Instruction Set: Arithmetic and Binary Logic Instructions.

14. Tech Help Manual. 8088/286/386/486/Pentium Instruction Set: Instruction Set Matrix.

15. Tech Help Manual. 8088/286/386/486/Pentium Instruction Set: Secondary Opcode Space.

16. NASM — The Netwide Assembler.