Математические основы информатики

Лекция 3.
Бинарная арифметика.

Сергей Леонидович Бабичев

Логика и числа.

Числа и их представление

- Для процессора числа представляются набором логических значений на множестве их n элементов.
- Мощность такого множества 2^n.
- Значение числа определяется соглашениями о кодировании.
- Кодирование чисел — биективное отображение множества чисел мощностью 2^n на множество кортежей $\{0,1\}^n$.
- Имеются различные способы кодирования множества чисел.
- $\mathbb{N}_{2^n} = S_u = [0 \dots 2^n)$

Знаковое кодирование

- Для кодирования чисел, имеющих знак имеются варианты:
  - Кодировать знак в одной из позиций. Остальные позиции кодируют абсолютную величину. *Прямой код.*
  - Все позиции кодируются инверсией. *Обратный код.*
  - Все позиции кодируются операцией вычитания числа из нуля. *Дополнительный код.*

Примеры знакового кодирования

Для $n = 8$:

x	10	-10	0
Прямой код	00001010	10001010	00000000 или 10000000
Обратный код	00001010	11110101	00000000 или 11111111
Дополнительный код	00001010	11110100	00000000

Перевод числа в дополнительный код

- Процессор оперирует числами в $\{0,1\}^n$.
- И операнды и результат есть элементы этого множества.
- При логических операциях конъюнкции, дизъюнкции, исключающего или отрицания размер результата совпадает с размеров операндов.
- При логических операциях сдвига и арифметических операциях результат может содержать более n разрядов.
- Все лишние разряды отбрасываются.
- В дальнейшем будем проводить действия в $\{0,1\}^n$, отображаемом на множество $\mathbb{N}_{2^n}$.

Реализация арифметических операций

- Пусть имеются логические элементы:
  - not с одним входом и одним выходом.
  - and с двумя входами и одним выходом.
  - or с двумя входами и одним выходом.

Определить операции сложения двух чисел при $n = 1$ (два раздельных входа) с получением результата $n = 2$ (два раздельных выхода).

Абсолютно все операции на компьютерах производятся на логических элементах.

Схема задачи.

Схема логических действий

x	y	r_1	r_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Схема логических действий

x	y	r_1	r_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$r_1 = x \wedge y$$

Коммутационная схема 1

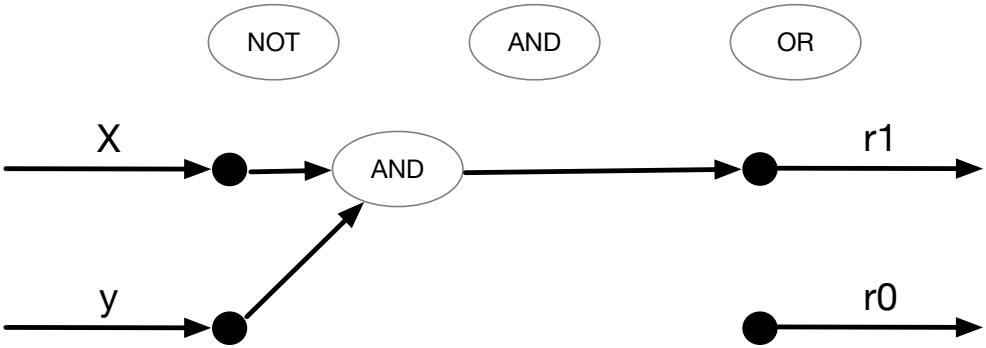


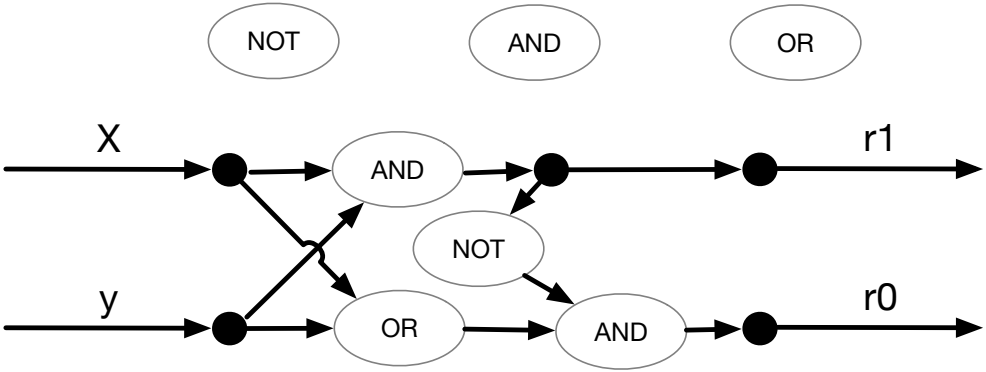
Схема логических действий

x	y	r_1	r_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$r_1 = x \wedge y$$

$$r_0 = x \oplus y = x \vee y \wedge \overline{x \wedge y}$$

Коммутационная схема 2



Дополнительный код

- Мы будем использовать наборы из фиксированного количества бит — $n = 8, 16, 32, 64$.
- Мы можем трактовать эти наборы как числа без знака. Тогда диапазон представления чисел будет $[0 \dots 2^n)$.
- Мы можем трактовать эти набора как числа со знаком. Тогда диапазон представления будет $[-2^{n-1} \dots 2^{n-1} - 1)$.

Клеточные автоматы

Клеточные автоматы

- Существует класс задач, называемый *клеточные автоматы*.
- Имеется *состояние* автомата, которое однозначно определяет следующее состояние.
- Все переходы состояний происходят одновременно.
- Переход между состояниями называется *совершением хода*.

Простой клеточный автомат

- Имеется строка из нулей и единиц, называемых *клетками*.
- Если в клетке находится единица, то она называется *живой*.
- Переход состояния клетки из нуля в единицу называется *рождением*.
- Переход состояния из единицы в ноль называется *смертью*.
- Каждая клетка, кроме самой левой и самой правой, имеет двух *соседей*.
- У самой левой клетки и самой правой клетки имеются фиктивные мёртвые соседи.

Простой клеточный автомат

- Правила перехода такие:
  1. Живая клетка, у которой оба соседа живых, умирает от перенаселения.
  2. Мёртвая клетка, у которой оба соседа живых воскресает.
  3. Живая клетка, у которой оба соседа мёртвых, умирает от одиночества.
  4. Мёртвая клетка, у которой оба соседа мёртвых, воскресает.
  5. В остальных случаях клетка состояние не изменяет.

Простой клеточный автомат: пример

- Пусть начальное состояние было таким:

1	0	1	1	0	0	0	1	0	1	0

Добавим фантомные клетки слева и справа:

0	1	0	1	1	0	0	0	1	0	1	0	0

Простой клеточный автомат: пример

- Пусть начальное состояние было таким:

1	0	1	1	0	0	0	1	0	1	0

Добавим фантомные клетки слева и справа:

0	1	0	1	1	0	0	0	1	0	1	0	0

Сделаем первый ход.

0	0	1	1	1	0	1	0	0	1	0	0	0

Простой клеточный автомат: пример

- Пусть начальное состояние было таким:

1	0	1	1	0	0	0	1	0	1	0

Добавим фантомные клетки слева и справа:

0	1	0	1	1	0	0	0	1	0	1	0	0

Сделаем первый ход.

0	0	1	1	1	0	1	0	0	1	0	0	0

После второго хода:

0	0	1	0	1	1	0	0	0	0	0	1	0

Где применяются клеточные автоматы

- Моделирования поведения толпы при выходе со стадиона.
- Моделирование решёточных газов.
- Моделирование транспортных потоков (Модель Нагеля-Шрекенберга)
- Моделирование популяций клеток и вирусов.
- ...

Зачем применять уравнения логики в клеточных автоматах?

- Для приемлемой точности нужны большие популяции клеток.
- Нужно много шагов моделирования.
- Последовательное моделирование медленное.
- Моделируют на многих вычислительных потоках и графических картах.
- Требуется создать модель, параллельно вычисляющую все новые состояния.

Как моделировать нашу задачу?

- Составим таблицу нового состояния клетки в зависимости от текущего состояния клетки и её соседей.
- Пусть клетки расположены так: ABC и мы хотим получить новое состояние клетки B'.

A	B	C	B'
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Решение задачи

A	B	C	B'
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

- Верхняя половина — $A = 0$. Справа — $B \equiv C$ или $\overline{B \oplus C}$.
- Нижняя половина — $A = 1$. Справа — $B \oplus C$.
- Итого: $B' = (\overline{A} \wedge \overline{B \oplus C}) \vee (A \wedge B \oplus C)$
- Упрощая: $B' = (\overline{A \vee B \oplus C}) \vee (A \wedge B \oplus C)$

Подсчёт количества бит в числе

Постановка задачи

- Имеется число x, представленное n битами.
- Требуется найти количество единичных битов.
- Имеется набор операций:
  - поразрядного или $\vee$, bitor;
  - поразрядного и $\wedge$, bitand;
  - поразрядного исключающего или $\oplus$, bitxor;
  - поразрядного отрицания (инверсии) not, bitnot;
  - поразрядного сдвига $shift\ left$, bitshl и $shift\ right$, bitshr.
- Результаты этих операций — новый набор из n битов, представляющий новое число.
- Операция истинности расширена на наборы как $b_0 \vee b_1 \vee \cdots \vee b_{n-1}$.
- Любое отличное от 0 число трактуется как истина.

Простое решение: просто считаем биты числа.

```
1: function COUNT1(x)
2:     count ← 0
3:     for all bit ∈ [0..n) do
4:         if x bitand (1 bitshl bit) then
5:             count ← count + 1
6:         end if
7:     end for
8:     return ← count
9: end function
```

Количество операций пропорционально n.

Альтернативный вариант

1: **function** COUNT2(x)
2: $count \leftarrow 0$
3: **for all** $bit \in [0..n)$ **do**
4: $count \leftarrow count + ((x \operatorname{bitshr} bit) \operatorname{bitand} 1)$
5: **end for**
6: $return \leftarrow count$
7: **end function**

Количество операций пропорционально n.

Ещё один альтернативный вариант

1: **function** COUNT3(x)
2: $count \leftarrow 0$
3: **for all** $bit \in [0..n)$ **do**
4: $count \leftarrow count + (x \operatorname{bitand} 1)$
5: $x \leftarrow x \operatorname{bitshr} 1$
6: **end for**
7: $return \leftarrow count$
8: **end function**

Количество операций пропорционально n.

Улучшения

- Как уменьшить количество операций?
- Использовать только единичные биты.
- Поэкспериментируем с операциями сложения и вычитания на 1 (инкремента и декремента).

Эксперимент с операцией инкремента

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$x+1$	0	0	1	0	1	1	0	1

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	1	1
$x+1$	0	0	1	1	0	0	0	0

	7	6	5	4	3	2	1	0
x	1	1	1	1	1	1	1	1
$x+1$	0	0	0	0	0	0	0	0

Эксперимент с операцией инкремента

Попробуем проделать операции bitor над парами.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$x+1$	0	0	1	0	1	1	0	1
$x \,\mathrm{bitor}(x+1)$	0	0	1	0	1	1	0	1

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	1	1
$x+1$	0	0	1	1	0	0	0	0
$x \,\mathrm{bitor}(x+1)$	0	0	1	1	1	1	1	1

	7	6	5	4	3	2	1	0
x	1	1	1	1	1	1	1	1
$x+1$	0	0	0	0	0	0	0	0
$x \,\mathrm{bitor}(x+1)$	1	1	1	1	1	1	1	1

Эксперимент с операцией инкремента

- Операция bitor числа с его инкрементом устанавливает самый правый из нулевых битов в 1.

Эксперимент с операцией декремента

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$x-1$	0	0	1	0	1	0	1	1

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	1	1
$x-1$	0	0	1	0	1	1	1	0

	7	6	5	4	3	2	1	0
x	0	0	0	0	0	0	0	0
$x-1$	1	1	1	1	1	1	1	1

Эксперимент с операцией декремента

Попробуем проделать операции bitand над парами.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$x-1$	0	0	1	0	1	0	1	1
$x \,\mathrm{bitand}(x-1)$	0	0	1	0	1	0	0	0

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	1	1
$x-1$	0	0	1	0	1	1	1	0
$x \,\mathrm{bitand}(x-1)$	0	0	1	0	1	1	1	0

	7	6	5	4	3	2	1	0
x	0	0	0	0	0	0	0	0
$x-1$	1	1	1	1	1	1	1	1
$x \,\mathrm{bitand}(x-1)$	0	0	0	0	0	0	0	0

Эксперимент с операцией декремента

- Операция bitand числа с его декрементом устанавливает самый правый единичный бит в 0.
- Как нам это поможет ускорить алгоритм в среднем?
- Будем уничтожать по одному биту до тех пор, пока число не обнулится.

Улучшенный вариант алгоритма

```
1: function COUNT4(x)
2:     count ← 0
3:     while x ≠ 0 do
4:         count ← count + 1
5:         x ← x bitand(x − 1)
6:     end while
7:     return ← count
8: end function
```

Количество операций пропорционально числу ненулевых битов в x.

Улучшенный вариант алгоритма

1: **function** COUNT4(x)
2: $count \leftarrow 0$
3: **while** $x \neq 0$ **do**
4: $count \leftarrow count + 1$
5: $x \leftarrow x \,\mathrm{bitand}(x - 1)$
6: **end while**
7: $return \leftarrow count$
8: **end function**

Количество операций пропорционально числу ненулевых битов в x.

Победа?

Попытка параллельности

- Мы исходим из предположения, что существуют элементарные операции, обрабатывающие всё множество одновременно.
- Вначале мы работали с единичными битами.
- Затем мы стали работать с группой бит.
- Попробуем работать со всеми битами сразу.
- Можно рассматривать 8 бит как одно дизъюнктивное множество в 8 бит.
- Можно рассматривать 8 бит как два дизъюнктивных множества по 4 бита.
- Можно рассматривать 8 бит как четыре дизъюнктивных множества по 2 бита.

- Как извлечь числовые представления этих множеств?

Извлечение числовых подмножеств: пример

- Рассмотрим число $x = 00101100$
- Чему равно число, представленное множеством бит $[2..4]$?
- Создадим *маску*, содержащую единицы в нужных позициях и нули в остальных.
- Проведём конъюнкцию x с маской.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$mask$	0	0	0	1	1	1	0	0
$x \operatorname{bitand} mask$	0	0	0	0	1	1	0	0

- Последний этап — сдвинуть результат вправо на два бита.

	7	6	5	4	3	2	1	0
$(x \operatorname{bitand} mask) \operatorname{bitshr} 2$	0	0	0	0	0	0	1	1

Извлечение числовых подмножеств

1: **function** EXTRACT($x, start, end$)
2: $length \leftarrow end - start + 1$
3: $mask \leftarrow (1 \text{ bitshl } length) - 1$ ▷ obtain length right ones
4: $return \leftarrow (x \text{ bitshr } start) \text{ bitand } mask$
5: **end function**

Считаем единичные биты: продолжение

- Рассмотрим x как 8 дизъюнктивных множеств (кортежей) длиной 1 бит.
- Задача: просуммировать представления этих множеств.
- Разобьём на чётные (красные) и нечётные (зелёные) элементы.
- Требуется сложить попарно соседние значения из красных и зелёных представлений.
- Результат сложения однозначных чисел не превосходит 10_2.
- Для хранения результата достаточно двух разрядов.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0

Считаем единичные биты: продолжение

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0

- Как получить четыре набора из красных значений?
- Наложив маску, удаляющую зелёные значения.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$mask_u$	0	1	0	1	0	1	0	1
$u \leftarrow x \text{ bitand } mask_u$	0	0	0	0	0	1	0	0

- Мы получили четыре дизъюнктивных множества по два элемента.
- Они представляют четыре двухбитных числа — $0, 0, 1, 0$.

Считаем единичные биты: продолжение

- Как теперь получить такие же четыре числа, порождённые зелёными элементами?
- Наложив маску, удаляющую красные значения.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$mask_v$	1	0	1	0	1	0	1	0
$x \text{ bitand } mask_v$	0	0	1	0	1	0	0	0
$v \leftarrow (x \text{ bitand } mask_v) \text{ bitshr } 1$	0	0	0	1	0	1	0	0

- Осталось сдвинуть результат вправо на 1 бит.
- Мы опять получили четыре дизъюнктивных множества по два элемента.
- Они представляют четыре двухбитных числа — $0, 1, 1, 0$.

Считаем единичные биты: продолжение

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
$u \leftarrow x \text{ bitand } mask_v$	0	0	0	0	0	1	0	0
$v \leftarrow (x \text{ bitand } mask_v) \text{ bitshr } 1$	0	0	0	1	0	1	0	0

- В четырёх 2-х разрядных кортежах содержатся счётчики количества бит для зелёных и красных половинок отдельно.
- Теперь осталось сложить вектора (четвёрки). Простой операцией +! Переполнения результатов сложения не произойдёт, так как они не превосходят двух и поместятся в двухразрядные числа.

	7	6	5	4	3	2	1	0
$x \leftarrow u + v$	0	0	0	1	1	0	0	0

Считаем единичные биты: продолжение

Повторим операцию, разбив вектор на чётные и нечётные элементы и сложив уже пары двухразрядных чисел. Опять чётные пары выделим красным цветом, нечётные — зелёным.

	7	6	5	4	3	2	1	0
x	0	0	0	1	1	0	0	0

Снова раскинем их по векторам — на сей раз это будут два 4-битных вектора и сложим:

	7	6	5	4	3	2	1	0
x	0	0	0	1	1	0	0	0
$mask_u$	0	0	1	1	0	0	1	1
$u \leftarrow x \,\text{bitand}\; mask_u$	0	0	0	1	0	0	0	0
$mask_v$	1	1	0	0	1	1	0	0
$v \leftarrow (x \,\text{bitand}\; mask_v)\,\text{bitshr}\,2$	0	0	0	0	0	0	1	0
$x \leftarrow u + v$	0	0	0	1	0	0	1	0

Считаем единичные биты: продолжение

Последгий этап: разбиение на два 4-х битных элемента и их сложение.

	7	6	5	4	3	2	1	0
x	0	0	0	1	0	0	1	0
$mask_u$	0	0	0	0	1	1	1	1
$u \leftarrow x \operatorname{bitand} mask_u$	0	0	0	1	0	0	1	0
$mask_v$	1	1	1	1	0	0	0	0
$v \leftarrow (x \operatorname{bitand} mask_v) \operatorname{bitshr} 4$	0	0	0	0	0	0	0	1
$x \leftarrow u + v$	0	0	0	0	0	0	1	1

Результат: x содержал 3 единичных бита.

Считаем единичные биты: продолжение

- Последний штрих: заметим, что $mask_v$ получается из $mask_u$ сдвигом на то же количество разрядов, что и сдвиг x для получения v.
- Можно считать v как $(x \operatorname{bitshr} l) \operatorname{bitand} mask_u$, где l — размер кортежа.

Считаем единичные биты: алгоритм

1: **function** $\textsc{count5}(x)$
2: $\quad x \leftarrow (x \,\text{bitand}\ 01010101_2) + ((x \,\text{bitshr}\ 1) \,\text{bitand}\ 01010101_2$
3: $\quad x \leftarrow (x \,\text{bitand}\ 00110011_2) + ((x \,\text{bitshr}\ 2) \,\text{bitand}\ 00110011_2$
4: $\quad x \leftarrow (x \,\text{bitand}\ 00001111_2) + ((x \,\text{bitshr}\ 4) \,\text{bitand}\ 00001111_2$
5: $\quad return \leftarrow x$
6: **end function**

Считаем единичные биты: результат

- Стал ли лучше алгоритм?
- Насколько каждое изменение улучшало результат?
- Вычисление по алгоритмам $COUNT$ для 32-битных последовательных чисел от 0 до 100000000:
- Компилятор: `gcc 10.2.1`.
- Ключи компиляции: `-O3`.
- Процессор: `Intel Xeon W3520`.
- Тактовая частота: `2.67 GHz`.

Алгоритм	Время, с
$COUNT1$	3.453
$COUNT3$	2.257
$COUNT4$	1.667
$COUNT5$	0.117

Так имеет ли смысл изучать битовые операции?