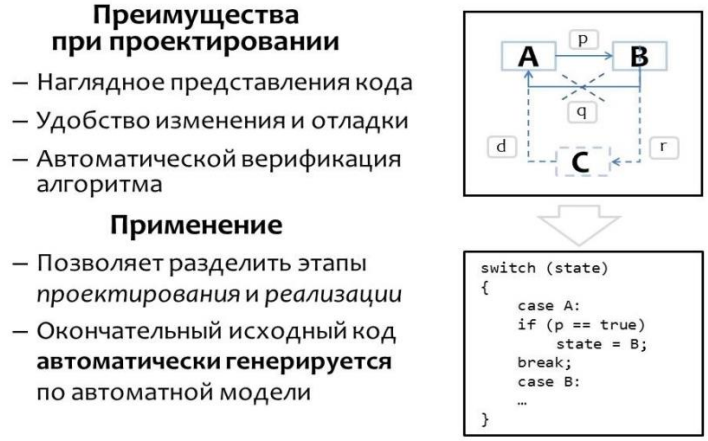
Транслятор описания конечного автомата в исходный код на языке описания аппаратуры Verilog

Автор: Пимкин Артем, 11 класс

Научный руководитель: Дединский Илья Рудольфович

1. Введение

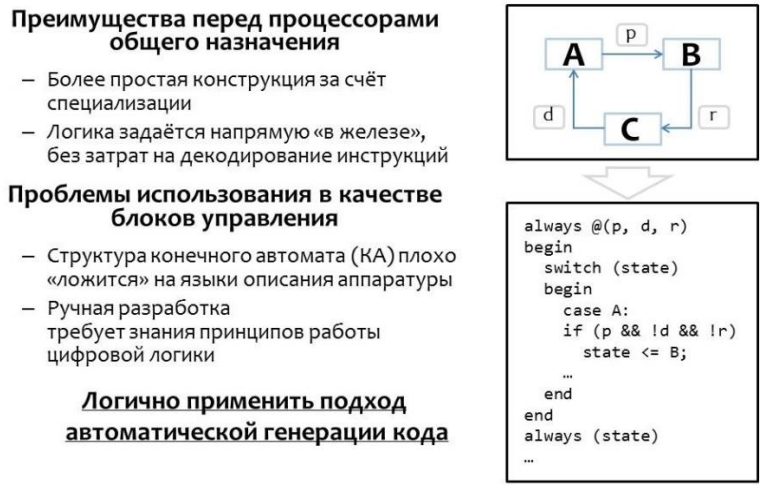
В последнее время все активнее развивается подход к алгоритмизации и программированию в целом, названный автоматным программированием [1]. Автоматное программирование основано на использовании конечных автоматов, описывающих поведение программ. С их помощью удобно проектировать управляющие и событийные системы. Одним из многих достоинств автоматного программирования является возможность формальной трансляции описания конечного автомата в исходные коды программ (см. рис. 1), что во многом упрощает проектирование программ использующих автоматный подход [2]. Также конечные автоматы могут верифицироваться [3](подвергаться автоматической проверке на корректность), что в совокупности с отсутствием человеческого фактора при трансляции позволяет полностью исключить появление ошибок при создании программы.



(а) (б)

Рис. 1. а) Преимущества и применение автоматного программирования.
б) Трансляция автоматной схемы в исходный текст программы.

Такой подход также возможно применить при проектировании аппаратного обеспечения, в том числе специализированных интегральных схем (ASIC). Более того, он активно применяется в различных задачах. Но использование данного подхода зачастую осложнено тем, что языки описания аппаратуры (например, Verilog [4]) не предназначены для описания конструкций подобного рода (см. рис. 2). Более того, они построены по совершенно другому принципу, нежели классические императивные языки, что ещё сильнее повышает сложность работы в этой области. Устранению обозначенных проблем и посвящена настоящая работа.



(а) (б)

Рис. 2. а) Использование ASIC в качестве блоков управления.
б) Трансляция автоматной схемы в описание на языке Verilog.

2. Цель и задачи работы

Целью работы является создание транслятора, способного преобразовать описание произвольного конечного автомата в исходный код на языке описания аппаратуры, и простого и понятного текстового языка описания конечных автоматов, адаптированного к таким описаниям. Это во многом упрощает использование автоматного подхода при проектировании аппаратуры. Транслятор генерирует по полученному описанию конечного автомата модуль-«черный ящик» на языке Verilog — абстрактную управляющую модель, поведение которой задано исходным автоматом. Это позволяет реализовать строго модульный подход к проектированию, так как управляющий модуль максимально абстрагирован от любого низкоуровневого кода. Более того, поскольку язык описания конечных автоматов не использует понятия, специфичные для разработки аппаратного обеспечения, появляется возможность разделить разработку управляющего алгоритма и аппаратного обеспечения между разными людьми. В частности, человек, разрабатывающий управляющий модуль, может не иметь никакого отношения к разработке остальных частей проекта и к программированию на языках описания аппаратуры. Для объемных проектов это может означать, например, более качественную работу, так как специалист высокого класса, умеющий разрабатывать высокоуровневые модели управления и специализирующийся на этом, выполнит работу по формализации алгоритма лучше, чем программист на языке Verilog, способный написать соответствующий код.

Таким образом, данная разработка может быть полезна как при создании больших, так и маленьких проектов из соображения качества работы, архитектуры и удобства в использовании, а так же за счет того, что данная разработка – бесплатный open source проект. Подходы к генерации аппаратуры на базе ПЛИС (FPGA) уже успешно применялись с использованием пакета Simulink для MathLab [5], однако это ПО не обладает открытым исходным кодом и требует дорогостоящей лицензии.

3. Язык описания конечных автоматов

Первой задачей является создание языка текстового описания конечных автоматов. Язык должен быть максимально простым и понятным для того, чтобы работать с ним мог человек, знающий лишь определение конечных автоматов. В настоящее время уже существуют языки описания конечных автоматов [6], но они не являются императивными, при том, что противоречит концепциям языков, так или иначе описывающих аппаратуру. Поэтому было решено разработать собственный язык описания конечных автоматов (далее SML — State Machine Language, см. рис. 3), максимально императивный.

Единственной декларативной конструкцией языка является определение перехода между состояниями; все остальные действия задаются императивно.

Рис. 3. Концепция языка SML.

Декларативность переходов между состояниями создаёт определённые проблемы при трансляции. В частности, язык Verilog является императивно-реактивным, что делает невозможным прямое копирование структуры конечного автомата. Данная проблема была решена путём введения двух независимо выполняющихся блоков, один из которых определяет следующее состояние, а второй — комбинацию выходных сигналов после перехода.

Стоит рассказать о принятой в SML модели автомата. Автомат представляет собой чёрный ящик, обладающий набором входных и выходных логических сигналов. Алгоритм, задаваемый автоматом, использует комбинации входных сигналов в качестве условий переходов и может изменять выходные сигналы в качестве действий.

В программе на языке описания конечных автоматов должно быть два основных блока: блок декларации всех состояний, входных и выходных сигналов и блок описания переходов между состояниями (см. рис. 4).

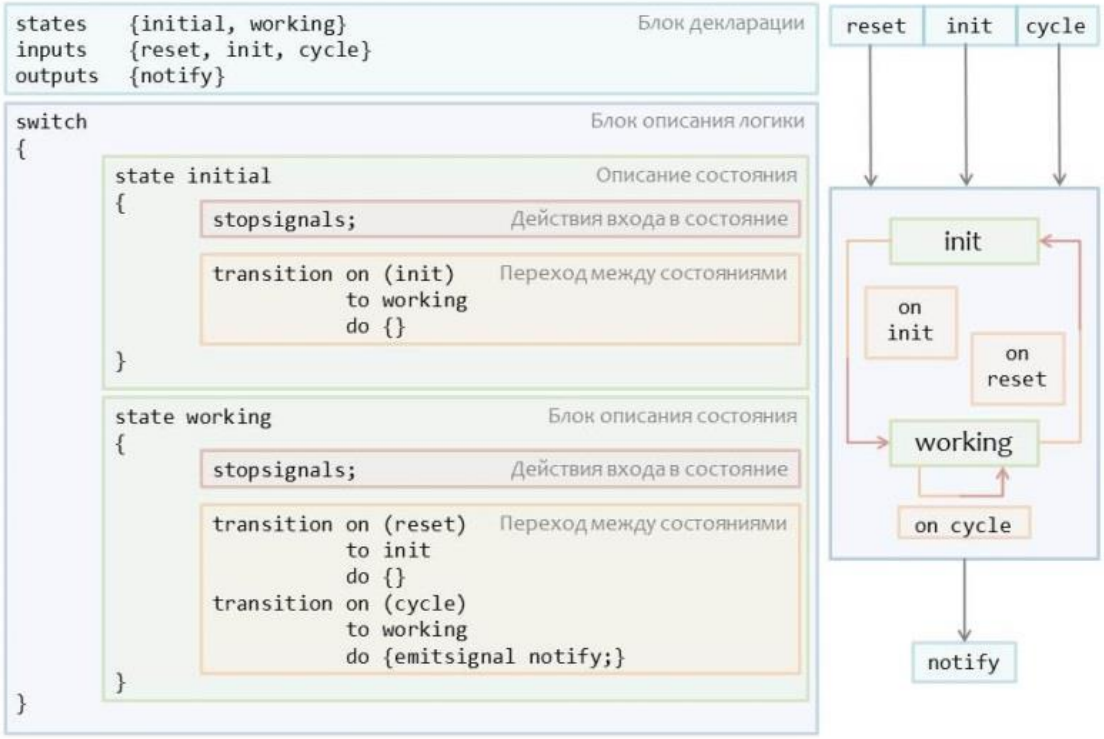


Рис. 4. Синтаксис языка SML.

3.1. Блок декларации

Каждому состоянию, входному или выходному сигналу сопоставляется его имя, по которому можно будет обращаться нему в основном блоке описания логики переходов. Это значит, что блок декларации содержит лишь перечисление всех имён состояний и сигналов. Формат перечисления приведен в листинге 1.

```
states  {state1, state2, state3}
inputs  {input1, input2, input3}
outputs {output1, output2, output3}
```

Листинг 1. Формат перечисления состояний, входных и выходных сигналов.

Состояние, которое обозначено первым в блоке `states`, будет считаться начальным.

3.2. Блок описания логики

Основной частью программы на SML является блок описания логики конечного автомата. Он содержит список переходов между состояниями автомата, каждый из которых включает в себя условие перехода и список побочных действий. Формат блока описания логики приведен в листинге 2.

```
switch
{
    state state1
    {
    }
    state state2
    {
    }
    state state3
    {
    }
}
```

Листинг 2. Формат описания состояний.

Каждый state-блок содержит описание одного состояния конечного автомата. Он включает в себя список (возможно условных) действий, выполняемых при переходе в данное состояние, и список переходов в другие состояния. Разберем подробнее каждую конструкцию.

Действия

- `emitsignal <signalName>;`
  Выставляет сигнал с именем `<signalName>` в единицу.
- `stopsignal <signalName>;`
  Выставляет сигнал с именем `<signalName>` в ноль.
- `stopsignals;`
  Выставляет все сигналы в ноль.
- `if (<inputs>) {<actions>}`
  Последовательность действий `<actions>` будет выполнена, если все входные сигналы, перечисленные через запятую на месте `<actions>`, выставлены в единицу.

Переходы

- `transition on (<inputs>) to <state> do {<actions>}`
  Переход в состояние с именем `<state>`, если все входные сигналы, перечисленные через запятую на месте `<inputs>`, выставлены в единицу. При этом при переходе будет выполнена последовательность действий, заданная на месте `<actions>`. Вложенные конструкции transition запрещены.

3.3. Пример SML-описания

Приведём пример описания на SML простейшего автомата, задающего модель поведения светофора (см. листинг 3).

```
states  {off, green, yellow, red}
inputs  {new_state, turn_off}
outputs {green_light, yellow_light, red_light}

switch
{
    state off
    {
         stopsignals;
         transition on (new_state) to green do {}
    }

    state green
    {
         stopsignals;
         emitsignal green_light;
         transition on (new_state) to yellow do {}
         transition on (turn_off)  to off      do {}

    }

    state yellow
    {
         stopsignals;
         emitsignal yellow_light;
         transition on (new_state) to red do {}
         transition on (turn_off)  to off do {}
    }

    state red
    {
         stopsignals;
         emitsignal red_light;
         transition on (new_state) to green do {}
         transition on (turn_off)  to off   do {}
    }
}
```

Листинг 3. Пример кода на SML, описывающего модель поведения светофора.

Этот пример работает следующим образом. У автомата есть четыре состояния:

- «Выключен»;
- «Горит красный»;
- «Горит желтый»;
- «Горит зеленый».

Начальным считается состояние «Выключен», так как оно первое в перечислении. Кроме того, у автомата есть два входных сигнала:

- «Перейти в новое состояние»;
- «Выключить»;

И три выходных состояния, ответственные за включение соответствующих индикаторов.

При переходе автомата в состояние «выключен» согласно описанию в соответствующем state-блоке производится отключение всех выходных сигналов (действие `stopsignals`). При подаче сигнала new_state автомат выполнит переход в состояние «горит зелёный», вследствие чего будет вклю-

чен сигнал зелёного индикатора. Из этого состояния возможны два перехода. В случае подачи сигнала `turn_off` автомат перейдёт в состояние «выключен»; если же будет подан сигнал `new_state`, автомат перейдёт в следующее состояние — «горит жёлтый». Остальные состояния работают аналогично.

4. Трансляция SML в Verilog

При разработке был использован модульно-конвейерный подход. Это означает, что за каждый этап обработки данных отвечает отдельный модуль, не зависящий от других. При этом выход каждого модуля является входом для следующего, а результат работы последнего модуля объявляется результатом работы всего транслятора. Такой подход позволяет легко изменять и отлаживать код, что делает данный проект платформой для реализации трансляторов языков описания автоматов в языки описания аппаратуры.

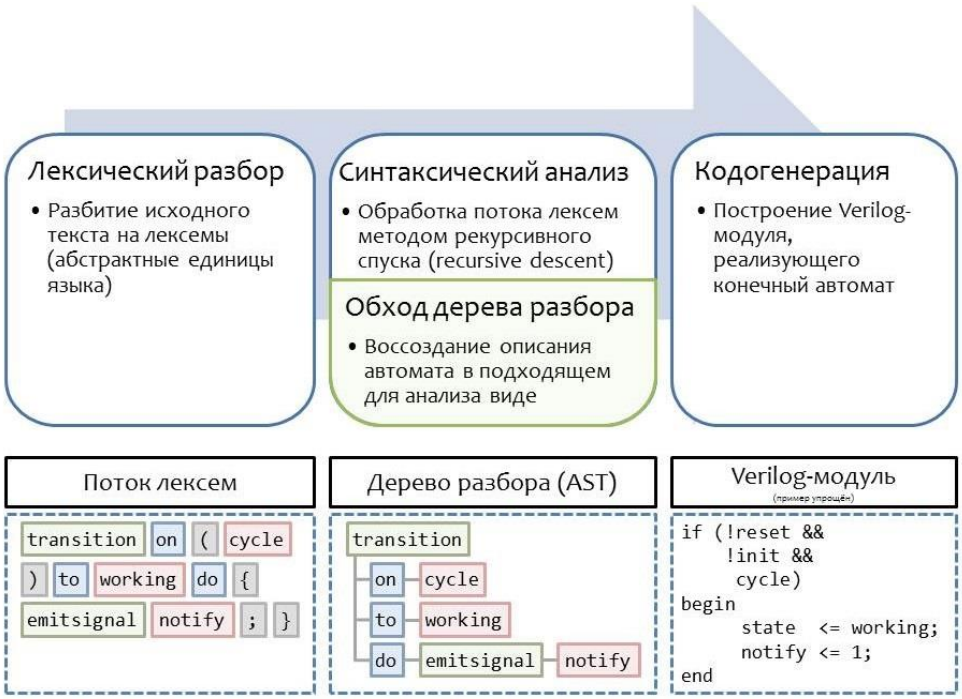


Рис. 5. Этапы трансляции SML в Verilog.

Трансляцию можно условно разбить на три этапа (см. рис. 5). Первым этапом будет являться разбитие входного потока на лексемы с сохранением их в память. Второй этап состоит из двух шагов: вначале по считанной последовательности лексем при помощи алгоритма рекурсивного спуска строится дерево разбора (AST), после чего выполняется обход дерева со сбором данных, необходимых для последующей кодогенерации (об этом позднее). Эти два шага объединены, то есть дерево в явном виде в память не сохраняется. Построение дерева, обход по нему и сбор данных происходят одновременно. Третьим этапом является генерация кода, то есть запись в выходной файл конструкций на языке Verilog, соответствующих описанию автомата.

Первый этап, а именно разбитие входного потока на лексемы, происходит с использованием заранее известного и легко изменяемого набора ключевых слов или символов языка. На этом этапе не происходит серьезной синтаксически-ориентированной обработки ошибок. Если считанная лексема не является зарезервированным для языка словом или символом, то ей автоматически присваивается тип «имя».

Второй этап является основным. Именно в нем происходит обработка входного потока и обработка ошибок. Если в процессе обработки входного потока будет обнаружена синтаксическая конструк-

ция, не являющаяся допустимой для SML, то происходит останов трансляции. В основе синтаксического анализа лежит алгоритм рекурсивного спуска (*recursive descent*).

Любое действие в описании на SML характеризуются небольшим количеством параметров, полностью его определяющих. В их число входят: название команды, аргументы (если есть), состояние, в котором происходит действие, состояние, при переходе в которое происходит действие (для команд в do-блоке конструкции transition), а также совокупность условий, при которых должно произойти действие. Для наглядности можно привести пример команды и контекст, в котором она встречается (см. листинг 4).

```
switch
{
    …
    state A
    {
        …
        if (a, b)
        {
            …
            if (c, d, e)
            {
                transition on (g) to B do {emitsignal x;}
            }
            …
        }
        …
    }
    …
}
```

Листинг 4. Пример контекста, в котором может встретиться команда.

Здесь действие оператора `emitsignal` можно полностью описать следующим образом: это команда `выставления логической единицы` с аргументом `x`, которая срабатывает в состоянии `A` при переходе в состояние `B`, при условии, что сигналы `a`, `b`, `c`, `d`, `e` и `g` выставлены в единицу. Это описание полностью задает как команду, так и результат ее трансляции в Verilog. Это означает, что по таким данным, собранным для всех встречаемых в описании на SML команд, можно сгенерировать необходимый код на Verilog. Генерация именно из такого формата одновременно является самой удобной, поскольку сбор данных сводится к набору небольших задач. Например, для того, чтобы найти совокупность условий, при которых команда будет выполнена, достаточно построить «родословную» для листа дерева, то есть всю цепочку узлов от корня до необходимого листа.

Третьим и последним этапом является генерация кода Verilog (см. рис. 6) из собранных при обходе по дереву данных. Здесь генерируется все необходимое для работы: декларация модуля, вспомогательные переменные, механизм отслеживания фронтов сигналов. После этого происходит генерация двух `always`-блоков. Один из них выполняется по положительному фронту тактового сигнала и отвечает за выполнение переходов между состояниями, а второй выполняется при смене состояния и отвечает за выполнение действий по входу в состояние.

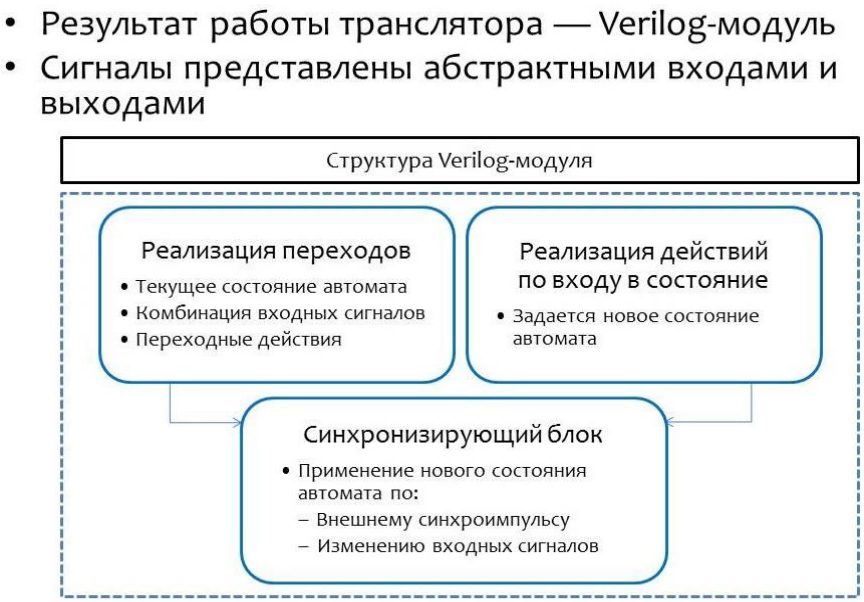


Рис. 6. Генерация Verilog-кода.

5. Проверка работоспособности генерируемого кода

Для проверки работоспособности была взята модель поведения светофора. Он является конечным автоматом. Нетрудно описать эту модель в виде схемы (см. рис. 7).

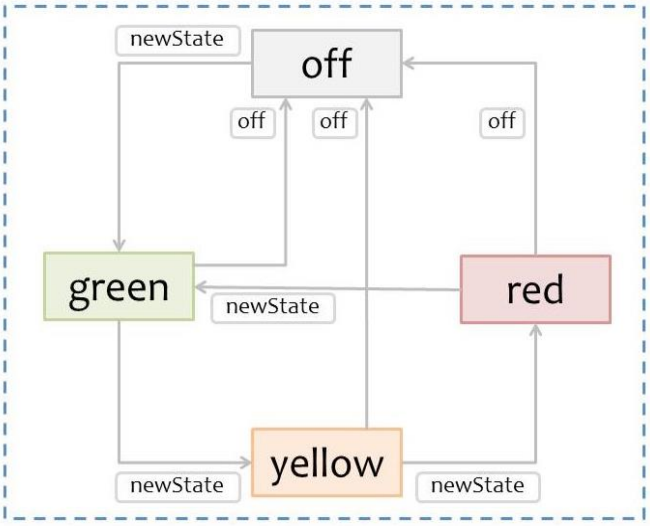


Рис. 7. Схема описания поведения светофора.

Эта схема может быть описана на языке SML, как было представлено ранее (см. листинг 3). После работы транслятора из текста на SML мы получаем код на Verilog, описывающий эту же модель. Размер полного листинга на Verilog слишком велик для помещения в статью, краткий фрагмент кода, полученного в результате трансляции можно увидеть на рис. 8.

```
module state_machine (input wire clk,
                      input wire inputs  [:0],
                      output reg outputs [2:0]);
… // Declaration of need variables and datas, generation of posedge
and negedge datas
always @(posedge clk)
begin
        … // Assignement of inputs
        case (state)
        0:
             begin
                     if ((inputs [2] && inputs [0]) &&
                         (posedge_inputs [0] || negedge_inputs [1] ||
                         posedge_inputs [2]))
                     begin
                             last_state <= state;
                             state = 1;
                     end
             end
        endcase
        …
end
```

Рис. 8. Фрагмент автоматически сгенерированного кода на Verilog.

Полученный код можно синтезировать в прошивку для FPGA. С помощью отладочной платы нетрудно проверить работоспособность полученного в результате трансляции из SML кода.

6. Заключение

Таким образом, в результате работы был реализован собственный текстовый язык описания конечных автоматов. Был разработан транслятор, способный по описанию конечного автомата автоматически генерировать код абстрактного управляющего модуля на языке Verilog. Испытана работоспособность генерируемого кода на отладочной плате с использованием FPGA.

Работа выполнена на языке C++, в среде KDevelop, в операционной системе Linux. Исходный код программы находится в репозитории `https://github.com/Nexx0f/SMT.` Размер исходного кода около 2000 LOC.

7. Литература

1. Н. И. Поликарпова, А. А. Шалыто. Автоматное программирование.
2. А.А.Шалыто, Н.И.Туккель, SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем, журнал "Программирование", №5, 2001 г.
3. С. Э. Вельдер, М. А. Лукин, А. А. Шалыто, Б. Р. Яминов, “Верификация автоматных программ”, “Наука”, 2011 г.
4. А.К. Поляков. Языки VHDL и Verilog в проектировании цифровой аппаратуры.
5. Ю. Ю. Янкин, А. А. Шалыто, “Автоматное программирование ПЛИС в задачах управления электроприводом”, журнал “Информационно-управляющие системы”, №1, 2011 г.
6. И.А. Лагунов, “Разработка текстового языка автоматного программирования ” (http://is.ifmo.ru/diploma-theses/fsml/)
7. Л.В. Столяров, публикация работы в журнале «Прикладная дискретная математика (Приложение)», Томск: ТомГУ, 2009, №1, С. 81-83.
8. П. Хоровиц. У. Хилл. Искусство схемотехники.
9. Материалы с сайта *stackoverflow.com.*