Разработка Системы Симулирования для Определения Резолюции π-тестов

Грицков С., Лазэр Д., Сорокин Г.
Технический Университет Молдовы
Кишинев, Молдова
gritscov@gmail.com

***Abstract* — This paper describes elaboration of a simulation system of pseudo-ring testing in C++ Builder. The elaborated application allows to simulate pseudo-ring testing with generation of different kinds of faults and calculate resolution of the tests.**

***Ключевые слова* — псевдо-кольцевое тестирование, симулирование, резолюция, неисправности.**

I. Введение

Любой тест обладает определенной резолюцией, определяющей список неисправностей, которые могут быть обнаружены данным тестом. При составлении маршевых тестов используют аналитический способ определения резолюции, то есть на основе последовательных умозаключений/шагов показывается, что определенный тип неисправности будет либо не будет полностью обнаружен [1]. Определить резолюцию π-тестов не всегда возможно аналитически. Решением данной задачи является проведение симуляции π-тестирования, при которой имитируется определенный тип неисправности в устройстве цифровой памяти. Симулирование в данном случае позволяет определить резолюцию любого теста, но требует дополнительных затрат на разработку самой системы симулирования [2].

II. Псевдо-Кольцевое Тестирование

Для реализации симулирования π-тестирования была разработана программа в среде C++ Builder. Разработанное приложение позволяет провести за доли секунды симуляцию теста с генерацией неисправностей определенного вида во всевозможных позициях, а реализация программы в среде C++ Builder в качестве приложения под Windows позволяет запускать ее пользователем без необходимости дополнительного программного обеспечения.

Система симулирования состоит из генератора неисправностей, устройства цифровой памяти, тестера и анализатора неисправностей. Данная система в виде блок-схемы представлена на рис. 1.

Тестер состоит из LFSR (linear feedback shift register – регистр сдвига с линейной обратной связью), который строится на основе неприводимого полинома $GF(2^4)$ $g(z)=1+2z+2z^2$ для 4-битной памяти. Данный LFSR и может быть построен на базе ресурсов тестируемой памяти.

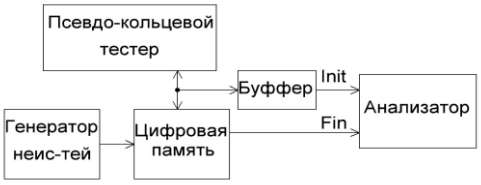

Рис. 1. Структурная схема системы симулирования π-тестирования.

В полиноме присутствует операция умножения на два над расширенным полем Галуа, что можно реализовать на основе оператора суммы по модулю 2 (рис. 2).

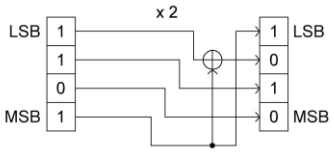

Рис. 2. Логическая схема операции умножения 4-битных чисел над расширенным полем Галуа.

Особенность LFSR на основе неприводимого полинома заключается в том, что в течение одного цикла регистр проходит все возможные комбинации без повторения и по завершении цикла возвращается в исходное состояние. Анализатор неисправностей сравнивает исходное значение (Init на рис. 1) LFSR с конечным (Fin на рис. 1), которые будут различными в случае возникновения неисправности.

Минимальный объем тестируемой памяти, при котором LFSR повторяет свое исходное значение для 4-битной памяти составляет 255 ячеек, что определяется периодом неприводимого полинома $GF(2^4)$ [2].

Для записи всевозможных значений в память процесс тестирования повторяется. Для 4-битной памяти минимальное число итераций равно 8.

III. Разработка Симулятора Для П-Тестирования

Независимо от типа симулируемой неисправности алгоритм функционирования системы симулирования π-тестирования представлен на рис. 2.

Алгоритм тестирования представляет собой три цикла: цикл по неисправностям (блок 2), внутри которого цикл по итерациям (блок 4) и цикл по ячейкам памяти (блок 6).

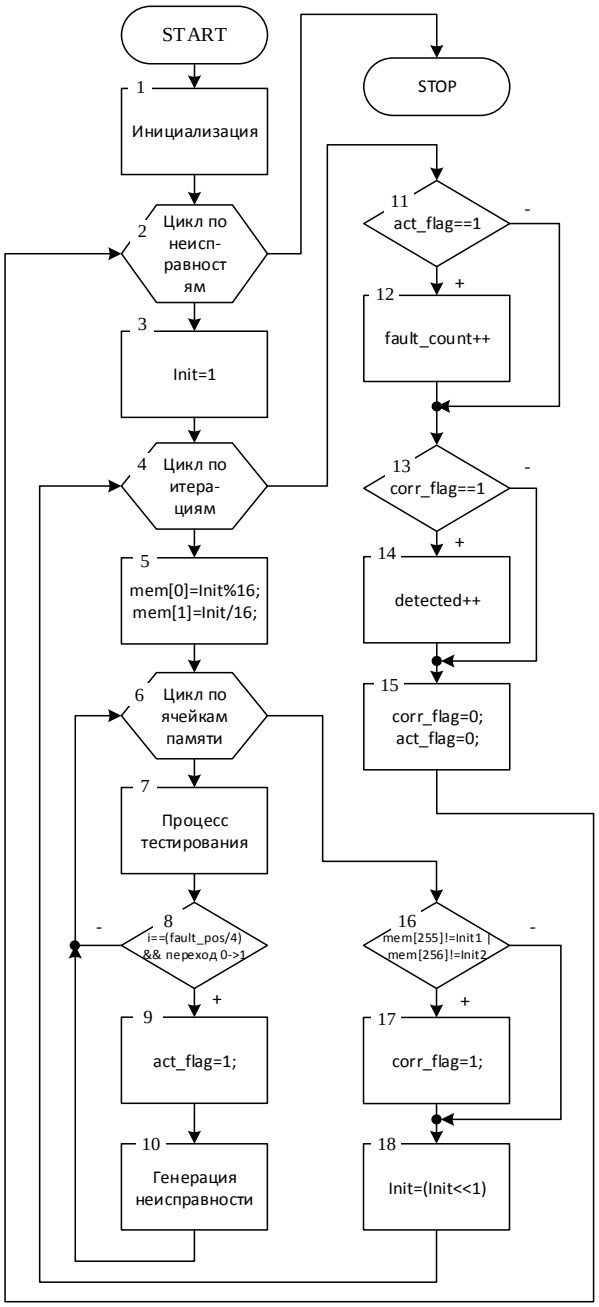

Рис. 3. Алгоритм процесса симулирования π-тестирования.

Тестирование проводится для памяти с имитацией одной неисправности в позиции fault_pos. Для этого выполняется полный набор итераций (их число зависит от разрядности памяти и модификации проводимого теста). Каждая итерация предполагает полное прохождение LFSR по всем ячейкам памяти. Итерации между собой отличаются различным значением исходного значения Init. Если хотя бы одна из итераций нашла неисправность (блок 16), то устанавливается флаг corr_flag (блок 17), а после цикла по итерациям инкрементируется счетчик найденных неисправностей (блок 14). Для 4-битной памяти число неисправностей составляет 1020: 255 ячеек по 4 бита. Внедрение неисправностей (блок 10) происходит внутри цикла по ячейкам памяти. Изначально проверяется счетчик неисправностей, который сравнивается с текущей позицией счетчика ячеек памяти (блок 8), а также при необходимости вносятся дополнительные условия в зависимости от типа неисправности.

Процесс тестирования происходит в результате прохождения LFSR по ячейкам памяти, поэтому значение i-той ячейки вычисляется согласно формуле:

mem[i]=Mult(mem[i-1])^Mult(mem[i-2]);

где символ ^ – операция суммы по модулю 2;

функция Mult() – функция вычисления результата умножения над расширенным полем Галуа значения ячейки памяти на 2. Для 4-битных данных умножение выполняется таким образом:

```
for (int i=0; i<4; i++)        // перевод 4-битного числа
{       arr[i+1]=A/p%2;        в массив 1-битных чисел
        p=p*2;   }
arr[0]=arr[4];                 // операция умножения
arr[1]=arr[1]^arr[4];
arr[4]=0;
```

где A – исходное число; arr[i] – число A, переведенное в массив из однобитных чисел. Имитация неисправности происходит следующим образом:

fault_mask=(1<<(fault_pos%4)); //маска для внедрения неисправности (принимает значения 0001, 0010, 0100, 1000). Внедрение неисправности:

константная s@1: mem[i]= mem[i] | fault_mask;

константная s@0: mem[i]= mem[i] & (15^fault_mask);

инверсная: mem[i-1]=(mem[i-1])^fault_mask;

Внедрение связанных неисправностей требует добавления дополнительного условия перехода определенного бита предыдущей ячейки памяти из 0 в 1 (или наоборот):

```
if ((i==(fault_pos/4)) &&  ((buff&fault_mask)==0)
  && ((mem[i]&fault_mask)==fault_mask) )
```

Заключение

Процесс тестирования 4-битной памяти с имитацией в ней неисправностей занимает 0,2-0,5с, а алгоритм позволяет модифицировать программу для расчета резолюции π-тестов для различных видов неисправностей. Процесс разработки системы симулирования в программной среде (язык С) требует в 5-10 раз меньше временных затрат по сравнению с процессом разработки данной системы на основе аппаратных ресурсов.

References

[1] С. В. Ярмолик, А. П. Занкович, А. А. Иванюк, "Маршевые тесты для самотестирования ОЗУ," монография, Минск, издательский центр БГУ, 2009, 270 с

[2] С. Грицков, Г. Сорокин, "Оценка качества псевдо-кольцевого тестирования устройств оперативной памяти," Conferinţa Studenţilor, Masteranzilor, Doctoranzilor şi Colaboratorilor, FRT, Chișinău, 2012.