Тестирование модулей арифметики с плавающей точкой микропроцессоров на соответствие стандарту IEEE 754

А.С. Камкин, М.М. Чупилко

Институт системного программирования РАН
109004, Москва, ул. Б. Коммунистическая, 25
E-mail: {kamkin, chupilko}@ispras.ru

Аннотация. В работе рассматривается методика функционального тестирования модулей арифметики с плавающей точкой микропроцессоров (FPUs, Floating Point Units) на соответствие стандарту IEEE 754. Методика основана на технологии тестирования UniTESK, но дополнена методами генерации тестов для операций над числами с плавающей точкой. Большое внимание в статье уделяется операциям деления и извлечения квадратного корня. Важной частью работы является описание опыта применения предлагаемой методики.

1. Введение

Модули арифметики с плавающей точкой (*FPUs*, *Floating Point Units*) являются важной составной частью микропроцессоров. Они используются в научно-технических расчетах и задачах обработки мультимедиа-данных, без них редко обходятся бортовые системы самолетов и космических спутников, они входят в состав станков с программным управлением и медицинского оборудования. Ошибки в таких системах могут стоить жизней или здоровья людей, поэтому не удивительно, что к модулям арифметики предъявляют очень строгие требования.

Большое влияние на формирование жестких требований оказывают также экономические факторы. В отличие от программного обеспечения, в котором исправление ошибки стоит сравнительно дешево, ошибка в аппаратном обеспечении, обнаруженная несвоевременно, может потребовать перевыпуск и замену продукции, а это сопряжено с очень высокими затратами. Так, известная ошибка в реализации инструкции деления чисел с плавающей точкой микропроцессора Pentium[1] [1] (FDIV bug) обошлась компании Intel в 475 миллионов долларов [2, 3].

Модули арифметики, как и любая цифровая аппаратура, разрабатываются на специальных *языках описания аппаратуры* (*HDLs*, *Hardware Description Languages*), например, Verilog или VHDL [4]. Такие языки позволяют описать функциональность электронной схемы, абстрагируясь от деталей расположения и соединения ее элементов. Использование языков описания аппаратуры значительно повышает продуктивность разработки аппаратуры, но не страхует от ошибок, поэтому *функциональное тестирование* по-прежнему остается актуальной и востребованной задачей.

---

1 Pentium — торговая марка нескольких поколений микропроцессоров семейства x86, выпускаемых компанией Intel с 22 марта 1993 года.

Формат и правила действий над числами с плавающей точкой определены в стандарте IEEE 754 [5] (он же — IEC 60559 [6]). Стандарт описывает операции сложения, умножения, вычитания, деления, вычисления остатка от деления, извлечения квадратного корня и преобразований между различными типами чисел — это базовый набор операций, реализуемых в FPU микропроцессоров. Для всех операций требуется, чтобы их результат получался из точного путем приведения к *представимому числу* согласно установленному *режиму округления*.

В идеале, чтобы убедиться в том, что операция над числами с плавающей точкой реализована корректно, необходимо проверить ее результат на всех возможных значениях операндов. Огромное пространство входных данных не позволяет это сделать за разумное время[2], поэтому необходимы методики построения тестов, которые при приемлемом размере тестового набора обеспечивают высокое качество тестирования.

В работе рассматривается методика построения тестов для FPU микропроцессоров, которая основана на технологии тестирования UniTESK [7], использующей формальные спецификации требований к системе для автоматизированного построения тестов на соответствие им [8]. Основные элементы технологии UniTESK применительно к модулям арифметики таковы. Требования на тестируемые операции представляются формально в виде эталонных реализаций. Чтобы сделать спецификации независимыми от конкретного интерфейса модуля, разрабатывается прослойка *медиаторов*, связывающих спецификации операций с тестируемым модулем.

Рассматриваемая методика дополняет технологию UniTESK методами генерации тестов для операций над числами с плавающей точкой. В рамках предлагаемого подхода используются несколько типов тестов: *сложные случаи округления*, *тесты на исключительные ситуации*, *особые случаи* (специальные значения операндов, граничные значения и другие) и *тесты с определенной битовой структурой операндов*. Подход был успешно апробирован в проектах по тестированию Verilog-моделей FPU, реализующих операции деления и извлечения квадратного корня.

Статья состоит из шести разделов, включая введение и заключение. Второй раздел посвящен стандарту IEEE 754. В этом разделе описываются основные положения стандарта касательно представления чисел с плавающей точкой, режимов округления и исключений, вызываемых операциями. В третьем разделе описаны предлагаемые методики построения тестов. Четвертый раздел описывает архитектуру разработанной тестовой системы, предназначенной для тестирования Verilog-моделей FPU. В пятом разделе приведены результаты апробации тестовой системы.

2. Обзор стандарта IEEE 754

Стандарт IEEE 754 определяет представление *двоичных чисел с плавающей точкой*, а также *режимы округления* операций и *исключения* — специальные

[2] Исключение составляют унарные операции над *числами одинарной точности*. (Определение чисел одинарной точности дается во втором разделе.)

флаги, сигнализирующие либо о некорректности значений операндов, либо об особенностях результата операции.

В дальнейшем будем называть двоичные числа с плавающей точкой просто *числами с плавающей точкой*, а действительные числа, точно представимые в виде чисел с плавающей точкой — *представимыми*.

2.1. Представление чисел с плавающей точкой

Согласно стандарту IEEE 754, битовое представление чисел с плавающей точкой содержит три составляющие: *знак*, *порядок* и *мантиссу*. Под знак отводится один бит (значение 0 соответствует положительным числам, 1 — отрицательным); число бит, отводимое под порядок и мантиссу, зависит от типа чисел.

Пусть под порядок отводится n бит. Число $B = 2^{n-1} - 1$ называется *смещением порядка* (*bias*). Знаковый бит S, порядок E и мантисса M числа x определяют его значение по следующим правилам: $x = (-1)^S \cdot 2^e \cdot m$, где:

- если $E \neq 0$, то $e = E - B$; иначе, $e = -B + 1$;
- если $0 < E < 2^n - 1$, то m имеет двоичное представление 1.M, то есть целая часть m равна 1, а последовательность цифр дробной части совпадает с последовательностью бит M; если $E = 0$, то m имеет двоичное представление 0.M, такие числа (с нулевым порядком) называются *денормализованными*.

Заметим, что среди чисел с плавающей точкой существует число –0, которое стандарт требует считать равным 0.

Стандартом определены особые комбинации значений порядка и мантиссы:

- *бесконечности* (±∞) — все разряды порядка равны единице, мантисса равна нулю ($E = 2^n - 1$, $M = 0$);
- *не-число* (*NaN*, *Not-a-Number*) — все разряды порядка равны единице, мантисса отлична от нуля ($E = 2^n - 1$, $M \neq 0$).

  Значения NaN подразделяются на *тихие* (*QNAN*, *Quiet NaN*) и *сигнальные* (*SNaN*, *Signaling NaN*), которые отличаются старшим битом мантиссы: если он равен единице, это QNaN; в противном случае — SNaN.

Стандарт IEEE 754 определяет несколько возможных типов чисел с плавающей точкой (отличающихся числом бит, отводимых под порядок и мантиссу), среди которых чаще всего используются *числа одинарной точности* (*singles*, *single precision numbers*) и *числа двойной точности* (*doubles*, *double precision numbers*).

Числа одинарной точности — это числа с плавающей точкой, для представления которых используется 32 бита: 1 бит на знак, 8 бит на порядок и 23 бита на мантиссу.

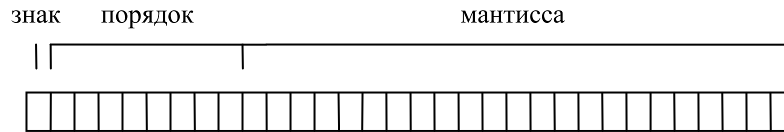


Рисунок 1. Представление числа одинарной точности.

Числа двойной точности — это числа с плавающей точкой, для представления которых используется 64 бита: 1 бит на знак, 11 бит на порядок и 52 бита на мантиссу.

2.2. Режимы округления

Стандарт IEEE 754 описывает операции сложения, умножения, вычитания, деления, вычисления остатка от деления, извлечения квадратного корня и преобразований между различными типами чисел. Общий принцип всех операций заключается в том, что результат получается из точного путем приведения к представимому числу согласно установленному режиму округления.

В стандарте определены четыре режима округления:

- *округление к ближайшему представимому числу*;
- *округление к* $-\infty$;
- *округление к* $+\infty$;
- *округление к 0*.

В режиме округления к ближайшему представимому числу, как следует из названия, результатом операции является представимое число, ближайшее к точному значению. Когда точное значение одинаково удалено от двух представимых чисел, выбирается то, у которого младший бит мантиссы равен нулю.

При округлении к $-\infty$ результатом является ближайшее представимое число, не превосходящее точного значения; при округлении к $+\infty$ — ближайшее представимое число, которое не меньше точного значения; при округлении к 0 — ближайшее представимое число, не превосходящее по абсолютной величине точного значения.

Три последних режима округления называют режимами *направленного округления* (*directed rounding*).

Для иллюстрации режимов округления рассмотрим примеры. В первом примере требуется округлить число с заданной мантиссой к ближайшему представимому числу одинарной точности:

- исходная мантисса: $10101010101010101010101\underline{011111111}_2$
- округленная мантисса: $10101010101010101010101\underline{000000000}_2$

В следующем примере требуется округлить число с заданной мантиссой в сторону 0:

- исходная мантисса: $10101010101010101010101\underline{111111111}_2$
- округленная мантисса: $10101010101010101010101\underline{000000000}_2$

2.3. Исключения

Стандартом IEEE 754 определены пять типов исключений:

- *invalid operation* — некорректная операция;
- *division by zero* — деление на нуль;
- *overflow* — переполнение;
- *underflow* — потеря значимости;
- *inexact* — потеря точности.

В стандарте приводится список ситуаций, в которых выставляется исключение *invalid operation*. Ниже представлены некоторые из них:

- любые операции над SNaN;
- сложение бесконечностей с разными знаками или вычитание бесконечностей с одинаковыми знаками;
- умножение $0 \cdot \infty$;
- деление $0/0$ или ∞/∞;
- взятие остатка от деления x *rem* y, где $y = 0$ или $x = \infty$;
- извлечение квадратного корня из отрицательного числа.

Рассмотрим остальные исключения на примере деления чисел двойной точности в режиме округления к 0.

Исключение *division by zero* выставляется, когда делитель равен нулю. Обязательное условие возникновения исключения — не равное нулю или бесконечности делимое. В случае возникновения исключения *division by zero* результат равен бесконечности:

делимое:	$7FEFFFFFFFFFFFFF_{16}$	(1.797693e+308)
делитель:	0000000000000000_{16}	(+0)
частное:	$7FF0000000000000_{16}$	($+\infty$)
исключения:	*division by zero*	

Исключение *overflow* возникает, когда результат операции превышает (по абсолютной величине) наибольшее представимое число:

делимое:	$7FEFFFFFFFFFFFFF_{16}$	(1.797693e+308)
делитель:	$0003A6B50B0F27BB_{16}$	(5.077619e–309)
частное:	$7FEFFFFFFFFFFFFF_{16}$	(1.797693e+308)
исключения:	*overflow* и *inexact*	

Исключение *underflow* возникает, когда результат операции настолько мал (по абсолютной величине), что он может быть представлен только денормализованным числом:

делимое:	0010000000000000_{16}	(2.225074e–308)
делитель:	$7FEFFFFFFFFFFFFF_{16}$	(1.797693e+308)
частное:	0000000000000000_{16}	(+0)
исключения:	*underflow* и *inexact*	

Исключение *inexact* возникает, когда результат операции отличается от точного значения:

делимое:	$27E83F0F3FFC9538_{16}$	(1.786431e+000)
делитель:	$FBAF43813FFFFFFF_{16}$	(2.000000e+000)
частное:	$2BC3037F3FEC9538_{16}$	(8.932153e–001)
исключения:	*inexact*	

3. Методика построения тестовых данных

Для качественного тестирования модулей арифметики с плавающей точкой тесты должны затрагивать различные аспекты их функциональности: выполнение операций, округление результата, выставление исключений и так далее. В связи с этим мы используем несколько типов тестов: *сложные случаи округления*, *тесты на исключительные ситуации*, *особые случаи* (специальные значения операндов, граничные значения и другие) и *тесты с определенной битовой структурой операндов*.

Помимо перечисленных типов тестов мы также используем *случайные тесты*, которые бывают полезны на ранних стадиях тестирования.

3.1. Сложные случаи округления

Сложным случаем округления (*hard-to-round case*, *extremal rounding boundary case*) называется ситуация, когда значения операндов таковы, что результат операции очень близок к числу, представимому в форме числа с плавающей точкой заданной точности, но отличается от него на величину значительно меньшую *единицы последнего разряда мантиссы* (*ulp*, *unit in the last place*). Очевидно, что значения операндов для сложных случаев округления зависят как от операции, так и от режима округления.

В сложных случаях округления возникает так называемая *дилемма составителя таблиц* (*table maker's dilemma*). Дилемма состоит в том, что для выбора правильно округленного числа с плавающей точкой следует вычислить много дополнительных бит мантиссы результата, значительно больше, чем имеется в рассматриваемом типе чисел с плавающей точкой [8].

Для более точного определения сложных случаев округления введем понятие *остаточных бит результата*. Под остаточными битами будем понимать последовательность битов точно вычисленного результата, начинающуюся с бита, следующего за младшим битом мантиссы. В общем случае последовательность остаточных бит может быть бесконечной.

Сложным случаем для режима округления к ближайшему представимому числу (*RN-hard*, *hard to round to nearest*) называется ситуация, когда остаточные биты результата начинаются на 100...0 или 011...1, где число нулей или, соответственно, единиц в конце последовательности достаточно велико.

Сложным случаем для режима направленного округления (*RD-hard*, *hard to round for directed rounding*) называется ситуация, когда остаточные биты результата начинаются на 00...01 или 11...10, где число нулей или, соответственно, единиц в начале последовательности достаточно велико.

Тематике генерации сложных случаев округления для различных операций, режимов округления и точности посвящено большое число исследований [9-16]. Используемый нами метод генерации сложных случаев округления для операции деления основан на работах [9-14]. Для операции извлечения квадратного корня мы использовали метод, описанный в работе [15].

Важно отметить, что тесты на сложные случаи округления для операции деления (по крайней мере, для режимов округления к $\pm\infty$) должны содержать как ситуации, в которых операнды имеют одинаковый знак, так и ситуации, в которых знаки операндов различны, поскольку знак результата (в этих режимах) влияет на округление. Такого рода ошибка была обнаружена нами в одной из реализаций FPU. Более подробное описание этой ошибки содержится в разделе "*Практическая апробация подхода*".

3.2. Исключительные ситуации

Как было сказано выше, стандарт IEEE 754 определяет пять типов исключений. Не каждая операция вызывает все перечисленные исключения, однако, если тестируемая операция может вызвать некоторое исключение, тесты, в которых соответствующие ситуации реализуются, должны обязательно входить в тестовый набор. Также в тестовый набор целесообразно включать тесты, в которых создаются ситуации "близкие" к исключительным, но таковыми не являющиеся.

Заметим, что для проверки правильности выставления исключения *inexact* можно использовать тесты на сложные случаи округления. Остановимся на операциях деления и извлечения квадратного корня. В силу своей природы операция извлечения квадратного корня никогда не вызывает исключений *overflow* и *underflow*. Тесты на исключение *division by zero* разрабатываются очевидным образом (они должны содержать деление конечных ненулевых чисел на ± 0). Рассмотрим подробнее тесты на исключение *invalid operation* (для обеих операций) и тесты на исключения *overflow* и *underflow* (для операции деления).

В стандарте IEEE 754 написано, что любая операция, использующая в качестве хотя бы одного операнда значение SNaN должна вызывать исключение *invalid operation*. Использование в качестве операндов SNaN и других особых значений реализуется в так называемых *тестах на особые значения*, которые будут описаны в соответствующем разделе.

Операция деления вызывает исключение *invalid operation* в случае неопределенности вида 0/0 или ∞/∞. Тесты для операции деления должны включать обе неопределенности с различными знаками операндов.

Операция извлечения квадратного корня вызывает исключение *invalid operation* в случае, если операнд является отрицательным числом (исключение составляет значение -0, для которого значение квадратного корня определено и равно -0). Тесты для операции извлечения квадратного корня должны включать отрицательные числа, включая -0.

Пусть M_1 и M_2 — мантиссы первого и второго операндов, E_1 и E_2 — их порядки. Обозначим через B смещение, а через E_{max} — максимальное значение порядка нормализованного числа[3]. Для наглядности рассмотрим случай, когда оба операнда являются нормализованными числами. Операция деления вызывает исключение *overflow*, если выполнено одно из следующих условий:

- $M_1 < M_2 \Rightarrow B + (E_1 - E_2) - 1 > E_{max}$:

  Если $M_1 < M_2$, результат деления мантисс лежит в интервале (0.5, 1). В этом случае для нормализации результата необходимо удвоить мантиссу и уменьшить порядок на единицу. Отсюда следует, что исключение возникает, если выполнено условие $B + (E_1 - E_2) - 1 > E_{max}$.

- $M_1 \geq M_2 \Rightarrow B + (E_1 - E_2) > E_{max}$:

  Если $M_1 \geq M_2$, результат деления мантисс лежит в полуинтервале [1, 2). В этом случае нормализация результата не требуется, поэтому условие на исключение имеет вид $B + (E_1 - E_2) > E_{max}$.

Пусть, как и прежде, M_1, M_2 — мантиссы первого и второго операндов, E_1 и E_2 — их порядки, B — смещение. Обозначим через E_{min} минимальное значение порядка нормализованного числа[4]. Для наглядности рассмотрим случай, когда оба операнда являются нормализованными числами. Исключение *underflow* возникает, если выполнено одно из следующих условий:

- $M_1 < M_2 \Rightarrow B + (E_1 - E_2) - 1 < E_{min}$;
- $M_1 \geq M_2 \Rightarrow B + (E_1 - E_2) < E_{min}$.

Тесты должны включать ситуации как первого, так и второго типа, а также ситуации "близкие" к ним. Такие ситуации могут быть получены ослаблением приведенных выше условий. Так, для исключения *overflow* можно использовать значение E_{max}, уменьшенное на единицу, а для исключения *underflow* — значение E_{min}, увеличенное на единицу.

3.3. Особые случаи

Под *особыми случаями* мы понимаем числа с плавающей точкой, которые имеют специальную, определенную стандартом IEEE 754 семантику (∞, NaN), либо числа, в которых порядок и/или мантисса имеют граничные значения. Основные особые случаи для чисел одинарной и двойной точности представлены в *таблице 1*.

[3] Значения B и E_{max} зависят от точности рассматриваемого типа чисел с плавающей точкой: для чисел ординарной точности B равно $7F_{16}$=127, E_{max} — FE_{16}=254; для чисел двойной точности — B равно $3FF_{16}$=1023, E_{max} — $7FE_{16}$=2046.

[4] E_{min} всегда равно 1 независимо от точности рассматриваемого типа чисел с плавающей точкой.

	одинарная точность			двойная точность		
	знак	**порядок**	**мантисса**	**знак**	**порядок**	**мантисса**
+0	0	00_{16}	0	0	000_{16}	0
–0	1	00_{16}	0	1	000_{16}	0
денорм. числа	{0,1}	00_{16}	любая ≠ 0	{0,1}	000_{16}	любая ≠ 0
+∞	0	FF_{16}	0	0	$7FF_{16}$	0
–∞	1	FF_{16}	0	1	$7FF_{16}$	0
SNaN	{0,1}	FF_{16}	любая ≠ 0, старший бит = 0	{0,1}	$7FF_{16}$	любая ≠ 0, старший бит = 0
QNaN	{0,1}	FF_{16}	любая ≠ 0, старший бит = 1	{0,1}	$7FF_{16}$	любая ≠ 0, старший бит = 1

Таблица 1. Основные особые случаи.

Граничные значения: наименьшие (по абсолютной величине) представимые числа и наибольшие (по абсолютной величине) представимые числа приведены в *таблице 2*.

	одинарная точность			двойная точность		
	знак	**порядок**	**мантисса**	**знак**	**порядок**	**мантисса**
наибольшее норм. число	{0,1}	FE_{16}	$7FFFFF_{16}$	{0,1}	$7FE_{16}$	$FF…F_{16}$
наименьшее норм. число	{0,1}	01_{16}	000000_{16}	{0,1}	001_{16}	$00...0_{16}$
наибольшее денорм. число	{0,1}	00_{16}	$7FFFFF_{16}$	{0,1}	000_{16}	$FF…F_{16}$
наименьшее денорм. число	{0,1}	00_{16}	000001_{16}	{0,1}	000_{16}	$00...1_{16}$

Таблица 2. Наибольшие и наименьшие представимые числа.

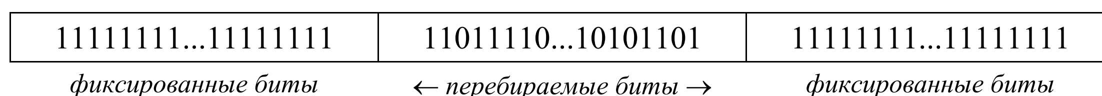
3.4. Числа с определенной битовой структурой

Использование чисел с определенной битовой структурой нацелено на тестирование следующих особенностей FPU микропроцессоров. Во-первых, для реализации операций над числами с плавающей точкой часто применяются *табличные алгоритмы*. Такие алгоритмы используют таблицу начальных приближений результата операции, которая индексируется с помощью определенных битов операндов. Поскольку таблица начальных приближений может содержать ошибочные данные, необходимо чтобы тесты покрывали все элементы этой таблицы. Во-вторых, результат операции часто вычисляется не полностью, а составляется из небольших частей путем сложения, конкатенации или с помощью других простых операций. В этом случае необходимо проверить правильность вычисления частей и правильность составления из них окончательного результата.

Рассмотрим известную ошибку в FPU микропроцессора Pentium (1994 г.), связанную с неправильными данными в таблице начальных приближений результата [1-3]. Тим Коу (Tim Coe) и Пинг Так Петер Танг (Ping Tak Peter Tang), а также Алан Эдельман (Alan Edelman) провели детальный анализ этой

ошибки [17, 18]. Они определили, что ошибка возникает, только в том случае, если мантисса делителя имеет вид $M = m_1m_2 \ldots$, где биты с 5-го по 10-ый включительно равны единице. Более того, биты $m_1m_2 \ldots m_4$ при этом должны принимать одно из пяти определенных значений. Вероятность возникновения подобных ошибок на случайных тестах очень мала. Более того, такие ошибки можно не обнаружить, используя тесты на сложные случаи округления или другие типы тестов, описанные выше.

В идеале, тестирование должно учитывать алгоритмы и схемы, использованные для реализации тестируемых операций. Если по каким-нибудь причинам это сделать не удается, например, когда не доступны исходные коды модуля и документация к нему, для перебора значений операндов (или только их мантисс) можно использовать следующую эвристику. В операндах полностью перебираются только N последовательных бит, значения остальных бит при этом остаются фиксированными. После полного перебора положение бит меняется. Процесс продолжается до тех пор, пока не будут использованы все возможные положения перебираемых бит. Число N должно быть не слишком маленьким, чтобы обеспечить приемлемое качество тестирования, и не слишком большим, чтобы позволить перебрать операнды (и их комбинации, если операндов несколько) за разумное время.



Подобную эвристику можно использовать и применительно к битам результата. В этом случае, чтобы получить значения операндов необходимо разрешить уравнение $f(x_1, \ldots, x_n) = y$, где f — тестируемая операция, y — требуемое значение результата, а $x_1, \ldots, x_n$ — искомые значения операндов.

Следует отметить, что использование таких простых эвристик позволило нам обнаружить и локализовать критическую ошибку в операции вычисления квадратного корня для чисел двойной точности в FPU промышленного микропроцессора. Более подробное описание этой ошибки содержится в разделе “*Практическая апробация подхода*”.

4. Архитектура тестовой системы

В соответствии с описанным подходом нами была разработана тестовая система для тестирования Verilog-моделей FPU микропроцессоров. Разработка велась на языке программирования C. В качестве основы построения тестовой системы мы использовали технологию UniTESK [7, 19]. Архитектура тестовой системы показана на *рисунке 2*.

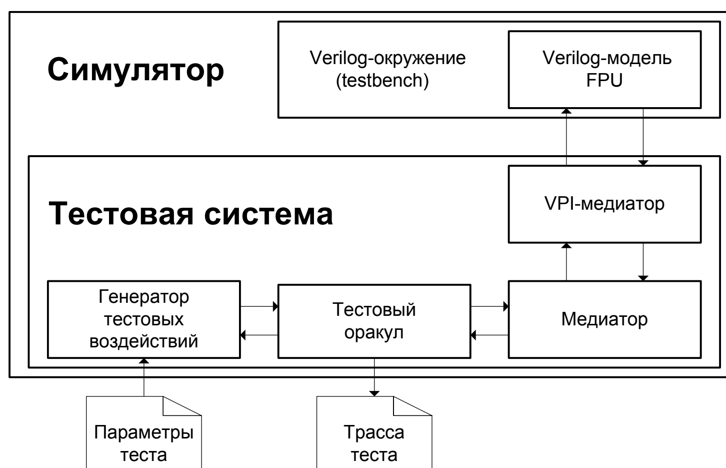


Рисунок 2. Архитектура тестовой системы.

Verilog-окружение (*testbench*) содержит экземпляр тестируемой Verilog-модели FPU. В начале тестирования окружение инициализирует тестируемую модель, после чего в цикле принимает от тестовой системы тестовые воздействия (код операции и значения операндов), подает их на тестируемую модель, ожидает реакции (результат операции и флаги исключений) и передает их тестовой системе для проверки. Прием тестовых воздействий и передача реакций осуществляются через VPI-медиатор.

VPI-медиатор связывает тестируемую модель с медиатором тестовой системы. Он реализует установку значений входных сигналов тестируемой модели в соответствии текущим тестовым воздействием, а также съем значений выходных сигналов. VPI-медиатор реализован с помощью интерфейса VPI (Verilog Procedural Interface) [20].

Медиатор преобразует значения операндов операции из внутреннего представления тестовой системы в представление, описанное в стандарте, и наоборот — значение результата операции из стандартного представления во внутреннее.

Тестовый оракул оценивает правильность поведения тестируемой модели в ответ на единичное тестовое воздействие. В основе тестового оракула лежат *эталонные реализации* операций над числами с плавающей точкой. Тестовый оракул записывает в *трассу теста* тестовые воздействия, на которых тестируемая модель FPU выдает ошибочный результат (результат, расходящийся с эталонной реализацией).

Генератор тестовых воздействий реализует построение тестов описанных выше типов: *сложные случаи округления*, *тесты на исключительные ситуации*, *особые случаи*, *тесты с определенной битовой структурой операндов* и *случайные тесты*. Выбор того или иного сценария генерации осуществляется на основе *параметров теста*, которые также задают точность чисел, режим округления и ограничения на число тестов.

4.1. Эталонные реализации операций

Для эталонной реализации операций используется *расширенное представление* чисел с плавающей точкой, в котором число бит, отводимое под мантиссу, достаточно для корректного округления результата реализуемых операций[5].

Эталонные реализации операций над числами с плавающей точкой описывают формально требования стандарта IEEE 754. Они вычисляют корректные результаты операций (с учетом установленного режима округления) и устанавливают флаги исключений. Рассмотрим кратко используемые нами эталонные реализации операций деления и извлечения квадратного корня без учета обработки исключительных ситуаций.

4.1.1. Эталонная реализация деления

Эталонная реализация операции деления основана на известном алгоритме деления двоичных чисел с фиксированной точкой. Перед использованием алгоритма вычисляется знак результата (исключающее ИЛИ знаков операндов) и порядок (разность порядков делимого и делителя). После этого операнды приводятся к специальному виду 01M, где M — это биты мантиссы.

Полученные значения операндов заносятся в переменные N (делимое) и D (делитель). Значения старших бит переменных будем называть *знаками*. Вычисление происходит последовательно от старшего к младшему биту мантиссы результата:

- если знак N равен нулю, в текущий разряд результата записывается единица, иначе — нуль;
- если знак N равен единице, N присваивается результат сложения N и D; иначе — результат сложения N и дополнения D;
- значение N сдвигается на один разряд влево.

Полученный результат преобразуется в форму мантиссы, для этого он сдвигается влево. Величиной сдвига является минимальное число, при котором старший единичный разряд оказывается за пределами разрядной сетки мантиссы. Заметим, что при сдвиге необходимо корректировать порядок результата.

4.1.2. Эталонная реализация извлечения квадратного корня

Извлечение квадратного корня реализовано следующим образом. Знак результата совпадает со знаком операнда (отрицательный знак операнда возможен только в случае, когда его значение равно –0), порядок равен половине порядка операнда[6]. Мантисса результата вычисляется с помощью

---

[5] Для корректного округления результата сложения, вычитания, умножения, деления и извлечения квадратного корня достаточно, чтобы размер расширенного представления мантиссы был равен $2 \cdot |M|$, где $|M|$ — это число бит, отводимое под мантиссу в рассматриваемом типе чисел с плавающей точкой [21].

[6] Если быть точнее, используются следующие правила вычислений:

- если $E < B$, то $E_{res} \leftarrow B - ((B - E) >> 1)$
  - если $E \mathit{mod}\ 2 = 0$, то $E_{res} \leftarrow E_{res} - 1$
- если $E \geq B$, то $E_{res} \leftarrow B + ((E - B) >> 1)$

алгоритма, в основе которого лежит возведение в квадрат. Последовательно от старшего к младшему биту мантиссы производятся следующие действия:

- текущий бит мантиссы результата устанавливается в единицу;
- полученная мантисса возводится в квадрат и сравнивается с исходным числом;
- если исходное число меньше, установленная единица сбрасывается в нуль.

5. Практическая апробация подхода

Разработанная тестовая система была успешно применена для тестировании двух различных Verilog-моделей FPU с одинаковым интерфейсом (будем обозначать их FPU_1 и FPU_2), реализующих операции деления и извлечения квадратного корня для чисел одинарной и двойной точности.

В результате тестирования FPU_1 была найдена критическая ошибка в реализации операции извлечения квадратного корня для чисел двойной точности. Ошибка была обнаружена на тестах с определенной битовой структурой операнда. Такие тесты позволили нам найти значения, на которых операция возвращала ошибочный результат. Анализ обнаруженных значений показал — если извлечь из них квадратный корень, то мантиссы результата имеют определенную структуру: младшие 36 бит являются единичными, а 16 старших — “случайные”. Мы перебрали все числа с описанной структурой (при некотором фиксированном порядке) — числа возводились в квадрат и подавались в качестве операнда. В результате было найдено большое число (около 6000 для каждого фиксированного порядка) “ошибочных” значений.

Рассмотрим примеры (округление осуществляется в сторону 0):

операнд:	$64300800FFFFFFFE_{16}$	(3.965019e+174)
полученный результат:	$52100403FFFFFFFF_{16}$	(1.991243e+087)
корректный результат:	$521003FFFFFFFFFF_{16}$	(1.991236e+087)
операнд:	$2CB01687E8FFFFFE_{16}$	(1.928163e–093)
полученный результат:	$36500B43FFFFFFFF_{16}$	(4,391102e–047)
корректный результат:	$36500B3FFFFFFFFF_{16}$	(4.391085e–047)

В результате тестирования FPU_2 были найдены ошибки, связанные с неточностями в округлении и выставлении флагов исключений. Например, была обнаружена ошибка, возникающая при округлении к $+\infty$ отрицательного результата (ошибка проявлялась как на числах одинарной точности, так и двойной).

Рассмотрим примеры:

делимое:	$68CDCD2C_{16}$	(7.774959e+024)
делитель:	$A8B5F04C_{16}$	(–2.019925e–014)
полученный результат:	$FF800000_{16}$	(–∞)
корректный результат:	$FF7FFFFF_{16}$	(–3.402823e+038)

делимое:	$983FFFFFBD727292_{16}$	(–7.013789e–192)
делитель:	$581000007B4947AD_{16}$	(1.576081e+116)
полученный результат:	$801FFFFEC6DFECA5_{16}$	(–4.450145e–308)
корректный результат:	$801FFFFEC6DFECA4_{16}$	(–4.450145e–308)

Тестовая система также использовалась для тестирования Verilog-модели модуля деления чисел одинарной точности, доступной на сайте [22]. Следует отметить, что адаптация тестовой системы под эту реализацию заняла около 30 минут. Изменения касались компонентов тестовой системы наиболее приближенных к тестируемой модели, а именно Verilog-окружения и VPI-медиатора (это около 10% от общего объема исходного кода тестовой системы). Ошибок обнаружено не было.

6. Заключение

В работе была рассмотрена методика функционального тестирования модулей арифметики с плавающей точкой микропроцессоров на соответствие стандарту IEEE 754. Методика основана на технологии тестирования UniTESK, но дополнена методами генерации тестов для операций над числами с плавающей точкой, которые позволяют осуществить разностороннее систематичное тестирование FPU. Разработанная согласно описанной методике тестовая система была использована для тестирования нескольких Verilog-моделей FPU, реализующих операции деления и извлечения квадратного корня для чисел одинарной и двойной точности. В результате тестирования было найдено несколько серьезных ошибок.

Литература

1. *Statistical Analysis of Floating Point Flaw in the Pentium Processor*. Intel Corporation, November 1994.
2. B. Beizer. *The Pentium Bug — An Industry Watershed*. Testing Techniques Newsletter (TTN), TTN Online Edition, September 1995.
3. A. Wolfe. *For Intel, It's a Case of FPU All Over Again*. EE Times, May 1997.
4. А.К. Поляков. *Языки VHDL и VERILOG в проектировании цифровой аппаратуры*. — М.: СОЛОН-Пресс, 2003.
5. *IEEE Standard for Binary Floating-Point Arithmetic 754-1985*, 1985.
6. IEC 60559:1989. *Binary Floating-Point Arithmetic for Microprocessor Systems*. Geneve: ISO, 1989.
7. http://www.unitesk.com.
8. В.В. Кулямин. *Формальные подходы к тестированию математических функций.* ИСП РАН, 2006.
9. D. Matula, L. McFearin. *Number Theoretic Foundations of Binary Floating Point Division with Rounding*. Real Numbers and Computers, 2000.

10. D. Matula, L. McFearin. *Generation and Analysis of Hard to Round Cases for Binary Floating Point Division*. Symposium on Computer Arithmetic, 2001.
11. D. Matula, L. McFearin. *Selecting a Well Distributed Hard Case Test Suite for IEEE Standard Floating Point Division*. International Conference on Computer Design, 2001.
12. D. Matula, L. McFearin. *A p × p Bit Fraction Model of Binary Floating Point Division and Extremal Rounding Cases*. Theoretical Computer Science, 2003.
13. D. Matula, L. McFearin. *A Formal Model and Efficient Traversal Algorithm for Generating Testbenches for Verification of IEEE Standard Floating Point Division*. Conference on Design, Automation and Test in Europe, 2006.
14. D. Matula, L. McFearin. *Generating a Benchmark Set of Extremal Rounding Boundary Instances for IEEE Standard Floating Point Division*. (http://engr.smu.edu/~matula/single/paper.ps)
15. W. Kahan. *A Test for Correctly Rounded Square Root*. Computer Science Department, Berkeley, 1994.
16. M. Parks. *Number-Theoretic Test Generation for Directed Rounding*. IEEE Transactions on Computers, Volume 49. 2000.
17. T. Coe, P.T.P. Tang. *It Takes Six Ones to Reach a Flaw*. Chinese University of Hong Kong. Technical Report 95-5(61), 1995.
18. A. Edelman. *The Mathematics of the Pentium Division Bug*. 1995. (http://www-math.mit.edu/~edelman/homepage/papers/pentiumbug.ps)
19. В.П. Иванников, А.С. Камкин, В.В. Кулямин, А.К. Петренко. *Применение технологии UniTesK для функционального тестирования моделей аппаратного обеспечения*. Препринт Института системного программирования РАН, 2005. (http://citforum.ru/SE/testing/unitesk_hard/)
20. S. Sutherland. *The Verilog PLI Handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface*. Springer, 2002.
21. D. Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Computing Surveys. March, 1991.
22. http://www.opencores.org.