Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Статическая верификация для языка HaSCoL

Курсовая работа студента 445 группы
Найданова Дмитрия Геннадьевича

Научный руководитель Медведев О. В.

Санкт-Петербург, 2012

Содержание

Введение

HaSCoL (Hardware - Software Codesign Language) — высокоуровневый язык описания аппаратуры, который разрабатывается на кафедре системного программирования. Язык поддерживает надежную и ненадежную доставку сообщений, неявную конвейеризацию обработчиков. Также язык поддерживает большое число конструкций обычных языков высокого уровня (циклы, массивы и т.д.). Код на хасколе транслируется в VHDL. Вместе с транслятором поставляется стандартный набор типов и операций над ними (bincompl). Также в коде на хасколе возможно использование внешних модулей на VHDL.

В разработке аппаратуры, статическая верификация — это формальное доказательство корректности системы. Такая формальная проверка нужна, если устройство используется в какой-то области, которая предъявляет особые требования к контролю качества (например, управление какими-то процессами на АЭС), так как тестирование не может обосновать, что система удовлетворяет определенному свойству или не содержит ошибок. Формальная верификация теоретически может быть полностью автоматизирована, поэтому под термином “формальная верификация” обычно подразумевают автоматическую верификацию с помощью какой-либо программы.

Один из способов верификации — это проверка модели (model checking). Он позволяет проверить, что в данной интерпретации заданная логическая формула является истинной (само название “проверка модели” появилось из логики, там моделью теории называется интерпретация, в которой истинны все формулы теории). Для верификации система и свойства, которые необходимо доказать, описываются каким-то формальным образом (для свойств это темпоральная логика, для системы — входной язык конкретного верификатора), а потом анализируется все пространство состояний системы. Например, если у нас есть два битовых регистра `ack1` и `ack2` , то

свойство “`ack1` и `ack2` никогда не будут равны 1 одновременно” можно записать так: `G !(ack1 & ack2)`. Для выполнения верификации при помощи проверки модели нужно построить для системы модель с адекватным числом состояний, представить ее на входном языке системы верификации, а также выразить подлежащие проверки свойства при помощи формул темпоральной логики, т.е. проверка корректности не может быть выполнена автоматически только по исходному коду системы.

Кроме проверки модели, есть другие подходы к верификации — с использованием формальной семантики языков программирования, логический вывод и другие, но они в рамках данной работы не рассматривались.

В рамках данной курсовой планируется, во-первых, верифицировать достаточно содержательное устройство, написанное на языке HaSCoL и, во-вторых, предложить какие-то рекомендации по улучшению языка, которые позволят генерировать из кода на языке HaSCoL более адекватную модель (под адекватностью подразумевается соответствие модели исходной системе и как можно большее снижение числа состояний модели).

1 Обзор существующих разработок

Можно найти достаточно много работ, посвященных формальной верификации, как по т.н. “verification-aware design”, так и описывающих опыт верификации конкретного устройства. Например,

“Verification-aware design” [6]. В статье кратко описываются подходы к верификации с использованием конкретного инструмента для проверки модели Cadence SMV. В качестве примера была выбрана модель процессора, поддерживающего 4 операции (`ALU, Branch, Load, Store`). Были сделаны некоторые предположения о том, как написать лучше подходящий для верификации дизайн, и проведены эксперименты, чтобы подтвердить их.

Подтвержденные предположения:

1. Нужно стараться разбивать систему на как можно меньшие модули.
2. Нужно избегать операций над частью симметричной переменной (над переменной, изменение значения которой заведомо никак не влияет на истинность проверяемых свойств).
3. Нужно сокращать глубину конвейеров и количество переменных, которые передаются от одной стадии к другой.

Опровергнутые предположения:

1. Не нужно выделять одного устройства из группы одинаковых (если, например, есть 4 одинаковых ALU, то не нужно у одного из них делать другой набор команд).
2. Нужно стараться передавать простые типы или как можно более простые структуры через входы и выходы модуля.

“Hardware Verification with Cadence SMV: A Multiplier Using Booth’s Algorithm” [4]. В статье описывается опыт верификации 4-битного умножителя, использующего алгоритм Бута, при помощи Cadence SMV. Указаны доказываемые свойства, подход к написанию модели, а также приведена статистическая информация от инструмента (время, затраченное на верификацию, количество проверенных состояний и т.п.).

“A methodology for system level Design for Verifiability” [2]. В работе изложены некоторые общие подходы, которые могут помочь улучшить дизайн для верификации, а также дан обзор диалекта VHDL, ориентированного на верификацию [3].

“Getting started with SMV” [7]. В руководстве пользователя для верификатора Cadence SMV в качестве примера была верифицирована модель процессора, использующая алгоритм Томасуло для внеочередного исполнения команд (out-of-order execution).

2 Реализация и верификация RISC процессора на языке SMV

Выбор инструмента для верификации. Так как проверка корректности достаточно сложной системы вручную почти невозможна, прежде всего необходимо выбрать программный инструмент для автоматической верификации.

Подробно были изучены два таких инструмента — Cadence SMV [1] и SPIN [5]. Cadence SMV был рассмотрен, т.к. его входной язык синтаксически достаточно похож на HaSCoL, а SPIN — потому что для него доступно много примеров и пособий. Оба бесплатны, у обоих есть достаточно подробная документация.

SPIN. Пакет SPIN, разработанный в исследовательском центре Bell Labs, позволяет построить модели программ (протоколов, драйверов, систем управления) и широкого класса дискретных систем, выразить требуемые свойства поведения на языке темпоральной логики и автоматически проверить выполнение этих свойств. Для построение модели используется язык Promela (само название SPIN расшифровывается как Simple Promela Interpreter).

Если верификатору удается доказать, что свойство системы не верно, то он выводит контрпример. Также кроме режима верификации доступен режим симуляции программы. Подробнее о SPIN можно прочесть в [10].

К сожалению, SPIN, в основном, рассчитан на верификацию параллельных программ, и создание систем с общим клоком, состоящих из более чем двух устройств, затруднительно.

Cadence SMV. Инструментарий для формальной верификации, разработанный в Cadence Berkeley Labs. Верификатор для описания свойств использует LTL. Модель может быть написана на специальном входном языке SMV или на диалекте верилога.

Если свойство не удается доказать, то также выводится контрпример.

В итоге, из-за существенного недостатка SPIN, был выбран Cadence SMV.

Верификация учебного процессора Nanoblaze. Процессор [9] был создан для того, чтобы давать задания на добавление к нему специфических инструкций для аппаратного ускорения конкретных программ. Процессор написан на хаскоде. Он является упрощенной реализацией архитектуры Xilinx Microblaze [8]. Список особенностей реализации таков:

1. Не поддержано никаких опциональных возможностей (прерывания, исключения, виртуальная память).

2. Не поддержано никаких специальных регистров — только 32 регистра общего назначения и бит carry.

3. В качестве памяти используется только двухпортовая статическая память на 4КБ с латентностью на любую операцию в 1 такт (Block RAM в FPGA от Xilinx).

4. Конвейер на две стадии, причем почти все декодирование и исполнение происходит в первой.

5. Процессор работает с 8-битными числами (первоначально было 32 бита, но размер был урезан для уменьшения количества состояний модели).

К процессору прилагаются все инструменты (ассемблер-линкер, gcc). Код процессора выложен в svn-репозитории http://code.google.com/p/hascolplayground/source

Изначально были решено, что саму память мы не верифицируем: отдельного модуля BRAM или регистров нет, мы проверяем только сам факт записи, блокировки на регистры и чтение, т.е. значения переменных, которые берутся из памяти (значения

входных сигналов хаскольного соответствующего обработчика), могут быть любыми. Это сделано потому что у 32 8-битных регистров может быть $2^{32*8} > 10^{76}$ состояний, а модель такого размера ни один верификатор не способен обработать. Также были убраны все отладочные флаги.

Почти вся логика работы процессора содержится в обработчике `local pipeline(...)`. Он был переписан в выполняющийся каждый такт код SMV `default {...} in {...}`. Для второй стадии обработчика была написана структура, которая содержит поля под все входные сигналы соответствующего обработчика и дополнительный флаг для того, чтобы определить, было ли послано сообщение.

Верификация процессора состояла из двух частей: доказательство того, что выполняются верно все вычисления, и доказательство правильности работы конвейера.

Корректность вычислений доказывалась следующим образом: любая выполняющая вычисления команда изменяет либо значение какого-то регистра, либо instruction pointer, значит нужно для всех поддерживаемых команд доказать свойства вида

$$if\ \ (operationode = add)\ \ valueToWrite := A + B$$

где valueToWrite — это либо значение, которое будет записано в регистр, либо новое значение instruction pointer.

Единственное, что делает вторая стадия - это запись в регистры. На первой стадии, если команда должна что-то записать, на соответствующий регистр выставляется блокировка, после записи блокировка снимается. Значит, корректность работы конвейера состоит из следующих свойств (`G` - темпоральный предикат, который значит "всегда"):

1. Всегда либо блокировок на регистр нет, либо выставлен флаг записи на второй стадии: `G (!lock.valid | writes2reg)`.

2. Чтение регистра, на который поставлена блокировка, не происходит никогда: `G (lock.valid -> lock.value != regWantToRead.value)`.

3. Если на первой стадии был выставлен флаг записи, то запись произойдет на второй стадии: `G(writes2reg -> X (regWantToWrite.valid))`.

4. Запись на второй стадии произойдет в тот регистр, на который захвачена блокировка: `G (lock.valid -> X (secondStageData.regWrite = lock.value))`.

3 Рекомендации к языку HaSCoL

Следующие вещи могли бы быть полезны в языке HaSCoL для получения более адекватной модели:

1. **Какой-нибудь способ указать, что над переменной будет проводиться лишь ограниченный набор операций.** В SMV некоторые типы оптимизаций вычислений зависят от типов переменных. Есть два специальных типа - `scalarset` и `ordset`.

   Тип является `scalarset`, если он симметричен, т.е. если можно заменить значение переменной такого типа на любое другое значение этого типа (пример — индекс в полностью ассоциативном кэше). Переменные такого типа почти не дают увеличения вычислительной сложности.

   Тип является `ordset`, если с переменными этого типа выполняются только следующие операции (пусть `x`, `y` - переменные такого типа, MAX - максимальное значение такого типа):

   - `x + 1`, `x - 1`
   - `x = 0`, `x = MAX`
   - `x < y`, `x > y`, `x >= y`, `x <= y`

   Для переменных такого типа возможно доказательство по индукции.

   Нужно уметь указывать, что переменные обладают такими свойствами, например:

   ```
   --@scalarset
   data indx : uint(12) = 0
   ```

```
--@ordset
data pc : uint(6) = 0
```

2. **Какой-нибудь способ указать, что данную переменную не нужно транслировать.** Некоторые переменные никак не влияют на модель, и поэтому не должны быть транслированы из исходного кода (например, те же отладочные флаги).

3. **Какой-нибудь способ указать, что некоторый оператор или метод можно считать неинтерпретированной функцией.** В некоторых случаях не важно, какую конкретно мы вызываем функцию, важно только, что мы используем один и тот же функциональный символ (например, когда мы не проверяем правильность вычислений, а только проверяем, что их результат всегда будет равен тем же вычислениям, выполненным в другом порядке). Вместо нее можно написать т.н. неинтерпретированную функцию, для которой известны только типы параметров и типы возвращаемого значения. Это существенно сократит количество состояний модели. Тогда в стандартную библиотеку bincompl можно добавить специальные операторы, которые при трансляции в VHDL будут заменяться соответствующими обычными операторами, а при генерации модели — неинтерпретированными функциями.

Заключение

Был проведен обзор существующих подходов к решению задачи статической верификации аппаратуры, в том числе опыт по верификации конкретных устройств. Был выбран и изучен конкретный верификатор Cadence SMV, и с помощью него верифицирован учебный процессор Nanoblazer. На основании полученного опыта, а также изучения алгоритмов, которые использует Cadence SMV, были предложены рекомендации по улучшению языка HaSCoL, которые помогут получить из исходного кода более адекватную модель.

Весь исходный код для Cadence SMV доступен в svn-репозитории
http://code.google.com/p/hascolplayground/

Список литературы

[1] Cadence SMV. Symbolic model checking tool. http://www.kenmcmil.com/smv.html

[2] Cagliero F., Camurati P., Colmanet A., Corno F., Cusinato M., Ferrero M., Prinetto P., Zaccaria C. A methodology for system level Design for Verifiability. Politecnico di Torino.

[3] Camurati P., Corno F., Prinetto P. VOVHDL: A verification-oriented dialect of VHDL. Politecnico di Torino.

[4] Hofstee K., Piegaze P. Hardware Verification with Cadence SMV: A Multiplier Using Booth's Algorithm. March, 2000.

[5] LTL model checking with SPIN. http://spinroot.com/spin/whatispin.html

[6] Lungu A., Sorin J. D. Verification-Aware Microprocessor Design. International Conference on Parallel Architectures and Compilation Techniques, Brasov, Romania, September, 2007.

[7] McMillan K.L. Getting started with SMV. Cadence Berkley Labs, 2001.

[8] MicroBlaze Processor Reference Guide. Xilinx, 2008. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[9] Описание учебного процессора Nanoblaze. http://code.google.com/p/hascolplayground/wiki/nanoblaze

[10] Шошмина И.В., Карпов Ю.Г. Введение в язык Promela и систему комплексной верификации SPIN. Санкт-Петербургский Государственный Политехнический университет, 2010.