Convolução como Caso de Estudo para Formalização em PVS da Correção Funcional de Implementações em FPGA's

Ariane Alves Almeida[1], **Jones Yudi Mori**[2], **Mauricio Ayala-Rincón**[1,3]

[1]Departamento de Ciência da Computação – Universidade de Brasília (UnB)

[2]Departamento de Engenharia Mecânica – Universidade de Brasília (UnB)

[3]Departamento de Matemática – Universidade de Brasília (UnB)

arianealves@aluno.unb.br, {ayala,jonesyudi}@unb.br

Resumo. *É apresentado um caso de estudo preliminar para verificação da correção funcional de operações algébricas implementadas em FPGA's. Tal caso é acerca de filtros de imagens baseados em convolução para tratamento de imagens digitais. A metodologia utilizada é baseada na equivalência funcional entre a definição matemática das operações e seu respectivo operador implementado em hardware. Foi utilizado o PVS como assistente de prova para facilitar a verificação proposta.*

1. Introdução

Com o objetivo de fornecer garantia de funcionamento adequado de um sistema, pode-se utilizar técnicas que o validem de modo sistemático, seja ele implementado em *software* ou *hardware*. Uma das formas de se obter tal garantia é por meio da verificação formal, que será aqui discutida.

Assim, o intuito deste trabalho é apresentar um caso de estudo que ilustra a Verificação por Equivalência Funcional como forma de verificar formalmente as propriedades funcionais de sistemas implementados em *hardware* com dispositivos reconfiguráveis. Para tal, utilizar-se-á o assistente de demonstração *Prototype Verification System* (PVS), a convolução para filtros de imagens e sua implementação em *Field Programmable Gate Arrays* (FPGA's).

A Seção 2 apresenta os conceitos referentes às arquiteturas reconfiguráveis e as arquiteturas FPGA; a Seção 3 as noções de formalização de sistemas e o assistente PVS; a Seção 4 explica como aplicar técnicas de verificação formal para comprovar matematicamente a correção funcional de sistemas implementados em FPGA's; a Seção 5 introduz um caso de estudo simples de verificação de um sistema implementado em FPGA; finalmente, a Seção 6 apresenta conclusões e trabalho futuro.

2. Sistemas Embarcados

Atualmente sistemas computacionais são concebidos através das mais diversas abordagens de *software*, *hardware* e *software-hardware*. Podendo estas duas últimas serem feitas através de dispositivos eletrônicos de propósitos específicos e arquitetura fixa, os *Application-Specific Integrated Circuits* (ASIC's), e também por abordagens híbridas que apresentam tanto características de *software* quanto de *hardware*, que é o caso dos dispositivos programáveis de propósito geral de arquitetura reconfigurável, como FPGAs ou *Complex Programmable Logic Device* (CPLD's) [Gaj and Chodowiec 2009].

2.1. FPGAs

Constituídos basicamente de blocos lógicos, registradores, blocos de entrada/saída e de tabelas de verdade, os FPGA's possibilitam sua programação através de linguagens de descrição de *hardware*, (similares às linguagens de programação de *software*). Isso fornece um nível de abstração muito maior ao projetista do que manipular diretamente elementos físicos para criar seu sistema e facilita o processo de desenho do circuito.

Um projeto em FPGA fornece uma velocidade muito maior quando comparado ao mesmo projeto concebido em *software* e executado em um processador de propósito geral, pois permite uma execução não sequencial e pode ser configurado para conter apenas as operações necessárias à execução do procedimento nele implantado, enquanto um circuito de propósito geral com instruções fixas, mesmo tendo um custo de produção mais baixo, necessita de todas as possíveis operações para qualquer estrutura de dados em seu projeto [Huffmire et al. 2010]. Outras vantagens observadas são o tempo e o custo de produção, que são bem inferiores aos da criação de um ASIC, e a possibilidade de reuso do circuito.

Ao finalizar um projeto em FPGA, é necessário verificar se suas funções estão realmente sendo realizadas e se estão executando corretamente, ou seja, se o resultado obtido é o esperado. Então, após a criação do sistema, é necessária sua validação.

Tal validação é obtida de diversas maneiras, sendo mais comumente utilizados os **testes funcionais**, que são simulações com diversos dados de entrada e comparação com os dados de saída esperados caso as funções tenham sido implementadas corretamente, e os **testes estruturais** ou **análise de *timing***, que levam em conta a estrutura interna do sistema criado e o *hardware* em que será inserido, como as conexões criadas e os possíveis atrasos dos sinais em determinada placa de *hardware*. Outra técnica menos difundida, mas que vem ganhando cada vez mais espaço e será melhor discutida na próxima Seção, é a **verificação formal** dos circuitos criados. Esse método pode ser aplicado em diversos estágios do desenvolvimento tanto de *software* quanto de *hardware* e sua aplicabilidade em sistemas FPGA será explicada na Seção 5.

3. Formalização de Sistemas

A verificação formal visa, por meio da aplicação de técnicas baseadas em formalismos matemáticos e lógicos, certificar que determinadas funcionalidades e propriedades dos sistemas e dados processados por estes sejam atendidas em qualquer situação. Além de ser uma abordagem matemática que visa especificar um sistema de maneira simples, completa e precisa para fornecer uma comunicação sem margens para ambiguidade aos projetistas, é uma tarefa muito complexa, bastante minunciosa e exaustiva, o que faz com que sua realização demande muito tempo.

A formalização se dá por meio da especificação do sistema e de provas de correção da sua funcionalidade. A fim de facilitar tal tarefa e evitar que sejam introduzidos erros oriundos de falha humana, foram criados os provadores de teorema, ou assistentes de prova, dos quais se destacam ACL2, Coq, HOL e PVS [Bernardo and Cimatti 2006].

3.1. PVS

Dentre diversos ambientes de especificação e formalização, foi escolhido o PVS, que inclui uma linguagem de especificação funcional com um sistema de tipos elaborado que

permite polimorfismo, subtipagem e tipos dependentes, e que tem uma linguagem de prova *à la Gentzen* com um sistema dedutivo de ordem superior, uma linguagem de especificação, um analisador léxico, um checador de tipos, bibliotecas de especificação e várias ferramentas de navegação, fornecendo um ambiente integrado para especificação e desenvolvimento de provas formais [Owre et al. 2001b].

A linguagem de especificação do sistema, mesmo não possuindo diversos recursos das linguagens de programação mais comuns, como laços de repetição e estruturas de dados sofisticadas, possibilita expressar em forma de conjecturas quais propriedades se espera que a especificação cumpra. Para isso, é necessário que quem está especificando um sistema tenha pleno conhecimento deste para conseguir representá-lo totalmente apenas com os recursos do PVS, que são seus tipos primitivos, a partir dos quais é possível criar outros, suas operações, estruturas condicionais e chamadas recursivas.

4. Formalização de Implementações em FPGA

A complexidade dos circuitos criados, sejam ASIC's ou FPGAs, tem aumentado a cada dia, ampliando a importância da verificação de seu correto funcionamento, o que chega a representar 80% do custo do projeto de um sistema [Drechsler 2004]. Assim sendo, vem aumentando o interesse nas técnicas de verificação formal, já que podem garantir corretude funcional de sistemas, o que não é possível com a tradicional análise dos resultados via testes e simulações [Perry and Foster 2005]. Por isso, é uma tarefa justificável e viável apesar do tempo e esforço demandados, especialmente para criação de sistemas críticos [Bernardo and Cimatti 2006].

4.1. Verificação de Equivalência Funcional

Existem basicamente duas maneiras de verificar formalmente um sistema, por checagem de equivalência (*Equivalence Checking* - EC) e por checagem de propriedades (*Property Checking* - PC). Com o método PC são identificadas e especificadas propriedades referentes ao circuito pretendido que são diretamente demonstradas ou testadas via técnicas como *Model Checking*. Tais propriedades então devem ser provadas satisfeitas em todas as possíveis circunstâncias do sistema [Kropf 1997], [Li and Thornton 2010].

Já a verificação por EC é uma técnica mais simples, que objetiva garantir a equivalência entre dois circuitos, que podem estar descritos de maneiras e níveis diferentes de abstração. Para isso, um dos circuitos a ser comparado deve ter sido previamente provado correto (por PC, por exemplo). Neste caso, deve-se estabelecer uma correspondência entre os dois e verificar então sua equivalência [Drechsler 2004], [Li and Thornton 2010].

Para este trabalho escolheu-se trabalhar com EC, supondo como circuito correto a especificação da definição matemática da função implementada sobre os FPGA's. Dessa maneira, assumindo a correção da especificação da definição matemática, pode-se analisar a arquitetura implementada comparando funcionalmente sua especificação com a da definição matemática. A técnica pode ser resumida como a seguir:

- Ambas as arquiteturas a serem comparadas devem ser especificadas de modo a deixar explícito e de maneira conservativa o que é funcionalmente e estruturalmente realizado sobre o FPGA em cada caso.
- Formaliza-se então um teorema de que as especificações são funcionalmente equivalentes, isto é, as mesmas entradas possuem saídas equivalentes. Sendo necessária para tal uma correspondência entre as entradas e objetos por elas manipulados.

5. Caso de Estudo Preliminar

Para ilustrar a aplicação da metodologia EC em PVS com implementação de FPGA, serão apresentadas partes de uma especificação criada para uma arquitetura de filtros de imagens baseados em convolução. Para tanto, serão apresentados alguns conceitos básicos dessa aplicação, bem como uma ilustração da arquitetura básica implementada em FPGA.

5.1. Filtros de Imagem Baseados em Convolução

A grande maioria das aplicações em FPGA's utilizam diversos operadores artiméticos para realizar suas funcionalidades. Dentre eles, no processamento de imagens digitais são utilizados filtros de imagem baseados em convolução, que ao serem desenvolvidos em FPGA's exploram o paralelismo a fim de melhorar o desempenho dessas aplicações.

A operação de convolução envolve uma imagem e os coeficientes do filtro a ser utilizado, assim sendo, dada uma imagem I com M linhas e N colunas, e um filtro (*kernel*) K, com $m \leq M$ linhas e $n \leq N$ colunas, a convolução entre I e K é definida pela Equação 1 [Pedrini and Schwartz 2007], onde $(K \otimes I)(x, y)$ representa o (x, y)-pixel da saída da aplicação do filtro, para $\frac{m}{2} \leq x \leq (M-1) - \frac{m}{2}$ e $\frac{n}{2} \leq y \leq (N-1) - \frac{n}{2}$. O filtro deve ter um tamanho tal que m e n sejam ímpares, de forma que o pixel resultante da aplicação do filtro seja aquele situado ao meio da vizinhança do filtro [Mori et al. 2011]. Logo, ao aplicar em toda a imagem esta equação, teremos como saída a imagem $(K \otimes I)$, que terá tamanho $M - m + 1 \times N - n + 1$.

$$(K \otimes I)(x, y) = \sum_{l=-\frac{m}{2}}^{\frac{m}{2}} \sum_{j=-\frac{n}{2}}^{\frac{n}{2}} K(l, j) * I(x + l, y + j) \tag{1}$$

As máscaras usadas como base para este trabalho são de tamanho fixo 3×3 (Equação 2), o que permite simplificar Equação 1, obtendo a Equação 3.

$$K = \begin{bmatrix} k1 & k2 & k3 \\ k4 & k5 & k6 \\ k7 & k8 & k9 \end{bmatrix} \tag{2}$$

$$(K \otimes I)(x, y) = \sum_{l=-1}^{1} \sum_{j=-1}^{1} K(l, j) * I(x + l, y + j) \tag{3}$$

A arquitetura básica da convolução aplicando um filtro de tamanho 3×3, mostrada na Figura 1, pode ser encontrada facilmente na literatura, como por exemplo em [Mori et al. 2011]. Os pixels representados em vermelho ilustram a vizinhança 3×3 na qual será aplicada a máscara, os amarelos ilustram um buffer para armazenamento temporário dos pixels a frente nas primeira e segunda linhas da imagem de entrada e os azuis representam o restante da imagem. O filtro é representado pelos pixels coloridos com os índices ki e o resultado pela saída do registrador verde com o símbolo de somatório.

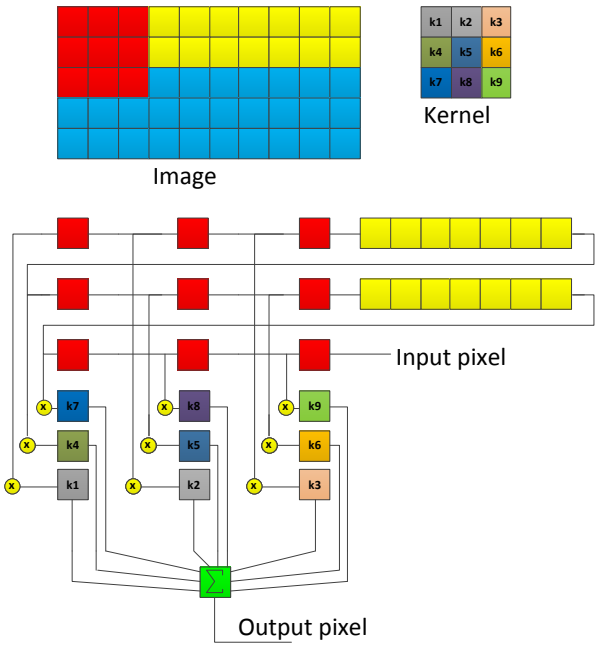

Figura 1. Arquitetura básica para uma convolução 3×3 por [Mori et al. 2011].

Vale ressaltar que a aplicação da definição matemática da convolução tem como resultado uma imagem $K \otimes I$ com $M - 2$ linhas e $N - 2$ colunas, ou seja, a imagem convoluída fica com 1 pixel a menos de borda que a imagem original. Por outro lado, a aplicação da convolução pela arquitetura apresentada gera um stream que representa uma imagem $K \otimes I$ com tamanho $M * N - 2 * N - 2$, ou seja, resulta em alguns pixels distorcidos a mais nas bordas laterais da imagem convoluída que devem ser desprezados.

5.2. Formalização da Arquitetura da Convolução

A fase de prova via EC entre a arquitetura da convolução para filtros de imagens em FPGA e sua equação matemática apresentadas na Seção 5.1 ainda está em andamento, logo, será apresentada apenas a parte de sua especificação em PVS segundo a metodologia da Seção 4 e os artifícios usados para tal.

A função especificada no Algoritmo 1 é relativa à aplicação da convolução em um pixel da imagem com o filtro em uma imagem de tamanho maior ou igual ao deste, e como estão sendo utilizados apenas filtros de tamanho 3×3, cada índice da matriz K é representado por uma constante ($k1$ a $k9$). Cada imagem `image` é representada por uma sequência finita de linhas de tamanho ≥ 3 (`rows`), e cada linha por sua vez é representada por uma sequência finita de inteiros de tamanho também ≥ 3 (`columns`).

Algoritmo 1 Função da Convolução Matemática em Um Pixel da Imagem.

```
loc_conv(im : image, i: posnat | i < rows - 1,
j : posnat | j < columns - 1 ) : int =
 im(i-1)(j-1)*k1 + im(i-1)(j)*k2 +  im(i-1)(j+1)*k3 +
 im(i)(j-1)*k4 + im(i)(j)*k5 + im(i)(j+1)*k6 +
 im(i+1)(j-1)*k7 + im(i+1)(j)*k8 +  im(i+1)(j+1)*k9
```

Esta função é então chamada pela função apresentada no Algoritmo 2, referente à aplicação da convolução, dada na Equação (3), a todos os pixels de entrada da imagem I, que retorna um elemento do tipo `conv`, que é uma sequência finita de linhas de tamanho $rows - 2$, e cada linha é uma sequência finita de inteiros de tamanho $columns - 2$, ou seja, o resultado final de $(K \otimes I)$.

Algoritmo 2 Função da Convolução Matemática em Toda uma Imagem.

```
convolution(im) : conv =
 (# length := rows - 2,
   seq := (LAMBDA (i: below[rows - 2]):
    (# length := columns - 2,
     seq := (LAMBDA (j : below[columns - 2]):
      loc_conv(im, i + 1,j + 1) ) #) ) #)
```

Alguns artifícios foram necessários para poder especificar a arquitetura da convolução. Neste caso, foi utilizada como entrada uma única sequência finita de inteiros `stream` de tamanho ≥ 9 para representar toda a imagem, já que é assim que esta é recebida pela arquitetura da Figura 1. Então, a manipulação de índices foi feita para simular uma matriz de pixels (imagem) e a passagem de todos seus pixels pelo buffer e sua posterior alocação na área vermelha, que é efetivamente a área de aplicação do filtro.

Respectivamente, nos Algoritmos 3 e 4, são especificadas a aplicação local da convolução em apenas uma vizinhança 3×3 descrita pela arquitetura e a aplicação da convolução em toda a imagem, utilizando-se da concatenação de sua chamada recursiva com a convolução local para chegar ao resultado final de $(K \otimes I)$.

Algoritmo 3 Função da Convolução Local em Um Pixel da Imagem pela Arquitetura

```
local_conv_arch(s: stream,  (n: numcolumns | n<= s`length),
(i: nat | i <= s`length - (n * 2) - 3 )) : int =
 s(i)*k1       + s(i+1)*k2       +   s(i+2)*k3 +
 s(i+n)*k4     + s(i+n+1)*k5     + s(i+n+2)*k6 +
 s(i+(n*2))*k7 + s(i+(n*2)+1)*k8 + s(i+(n*2)+2)*k9
```

Algoritmo 4 Função de Aplicação da Convolução a Todos os Pixels de uma Imagem

```
convolution_arc(s: stream, n: numcolumns,
(i: nat | i <= s`length - (n * 2) - 3 )):
RECURSIVE finite_sequence[int] =
 IF i = s`length - ((n * 2) + 3)  THEN
  add_first(local_conv_arch(s,n,i), empty_seq)
 ELSE
  add_first(local_conv_arch(s,n,i), convolution_arc(s,n,i+1))
 ENDIF
 MEASURE (s`length - i)
```

Para formalizar a equivalência funcional entre a arquitetura e a equação, é necessário que haja uma correspondência entre suas entradas, como visto na Seção 4.1. Como os elementos de entrada da definição matemática (sequência de sequências) e da arquitetura (sequência de inteiros) são diferentes, os elementos de entrada da primeira são transformados naqueles aceitos como entrada pela segunda através da função especificada no Algoritmo 5 com uso de chamadas recursivas.

Algoritmo 5 Função que Transforma uma Imagem (Sequência de Sequências) em uma Sequência Única.

```
image_to_seq (im: image, i: below[im`length-2]): RECURSIVE stream=
 IF i = im`length-3 THEN
  im(im`length-3) o im(im`length-2) o im(im`length-1)
 ELSE
  im(i) o image_to_seq(im,i+1)
 ENDIF
MEASURE (im`length-i)
```

A partir disso pode-se especificar um lema que expresse a equivalência funcional entre a equação da convolução e a arquitetura ilustrada na Figura 1, através da correspondência entre os elementos do resultado de quando aplicadas a equação e a implementação da arquitetura, sendo necessário desprezar os elementos da borda da imagem gerada pela arquitetura, pois seus resultados apresentam uma borda extra distorcida em relação ao resultado gerado pela equação matemática, como mencionado anteriormente.

O lema do Algoritmo 6 expressa essa correspondência entre os elementos resultantes das duas especificações. Para demonstrá-lo é necessária uma prova por indução aninhada sobre os índices `i` e `j`, pois ao percorrer a imagem `im`, tanto para aplicação das funções quanto para relacionar os elementos dos resultados, estes índices se incrementam, decrementando o tamanho da imagem restante a ser analisada.

Algoritmo 6 Lemma da Equivalência Funcional da Convolução Matemática com sua Arquitetura em FPGA

```
conv_arc_is_same_math : LEMMA
  FORALL (im: image, i:below[rows - 2], j:below[columns - 2]) :
    convolution(im)(i)(j) =
    convolution_arc(image_to_seq(im,0), columns, 0)(i*columns + j)
```

Além disso, é necessário o uso de lemas auxiliares que relacionem previamente os elementos de uma imagem qualquer com seu stream resultante da aplicação da função `image_to_seq` especificada acima e que determinem exatamente o tamanho do resultado da arquitetura da convolução aplicada à um stream, já que isso não é explícito na definição da função e é de extrema importância durante a prova.

Quando concluída a prova do lema acima, dado que a equação matemática está correta por hipótese, ficará provado que esta arquitetura também é funcionalmente correta.

6. Conclusão

Foi apresentada a importância da verificação formal para garantir a corretude em sistemas implantados em FPGA, que podem ser utilizados como protótipo para a criação de ASIC's corretos, já que estes últimos são mais caros para se produzir e qualquer erro de projeto pode aumentar ainda mais este custo.

O caso de estudo aqui apresentado, além de ilustrar o uso de EC para formalização de sistemas com operadores matemáticos implementados em FPGA's, explana a diferença entre as especificações de uma definição matemática e de sua implementação em um FPGA, mostrando a necessidade do uso de artifícios e manipulações para relacionar os resultados obtidos pelos dois e provar então a corretude funcional dessa arquitetura.

Assim sendo, pode-se utilizar a prova de corretude por equivalência funcional do sistema aqui apresentado, bem como de outros operadores aritméticos para criação e verificação de outros sistemas mais complexos que tenham tais funcionalidades definidas como parte do circuito, a fim de garantir sua corretude.

Referências

Bernardo, M. and Cimatti, A. (2006). *Formal Methods for Hardware Verification: 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, 2006, Advances Lectures*. Lecture Notes in Computer Science / Programming and Software Engineering. Springer.

Drechsler, R. (2004). *Advanced Formal Verification*. Falk Symposium Series. Springer.

Gaj, K. and Chodowiec, P. (2009). FPGA and ASIC Implementations of AES. In Çetin Kaya Koç, editor, *Cryptographic Engeneering*. Springer, New York, NY - Estados Unidos.

Huffmire, T., Nguyen, C. I. T. D., Levin, T., Kastner, R., and Sherwood, T. (2010). *Handbook of FPGA Design Security*. Springer.

Kropf, T. (1997). *Formal Hardware Verification: Methods and Systems in Comparison*. Lecture notes in computer science. Springer.

Li, L. and Thornton, M. (2010). *Digital System Verification: A Combined Formal Methods and Simulation Framework*. Synthesis Lectures on Digital Circuits and Systems Series. Morgan & Claypool.

Mori, J., Sanchez-Ferreira, C., Muñoz, D., Llanos, C., and Berger, P. (2011). An unified approach for convolution-based image filtering on reconfigurable systems. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 63 –68.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (2001a). *PVS Language Reference*. NASA.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (2001b). *PVS System Guide*. NASA.

Pedrini, H. and Schwartz, W. R. (2007). *Análise de Imagens Digitais: Princípios, Algoritmos e Aplicações*. Editora Thomson Learning.

Perry, D. and Foster, H. (2005). *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill Electronic engineering. McGraw-Hill Companies,Incorporated.