Utilização da Tecnologia XML no Desenvolvimento de Arquitecturas Específicas

Rui Rodrigues[1], Ricardo Ferreira[2], João M. P. Cardoso[1,3]

[1] Universidade do Algarve, Campus de Gambelas, 8000-117, Faro, Portugal
[2] Departamento de Informática, Universidade Federal de Viçosa,
Viçosa 36570 000, Brazil
[3] INESC-ID, 1000-029, Lisboa, Portugal
{rrodrigues@ualg.pt, cacau@dpi.ufv.br , jmpc@acm.org}

Resumo. Este artigo apresenta dois exemplos da utilização de XML na investigação e desenvolvimento de arquitecturas computacionais específicas. A tecnologia XML tem sido utilizada pelos autores deste artigo para descrever estruturas computacionais, e como meta-linguagem para criação de dialectos XML que facilitam a especificação de determinadas funcionalidades. Transformadores XSL têm sido utilizados para a geração de descrições em Java, VHDL, etc. As experiências têm mostrado muitas vantagens na utilização da tecnologia XML nos contextos abordados. O artigo ilustra a utilização de XML, foca as vantagens principais, e comenta as facilidades consideradas mais importantes.

1. Introdução

A utilização de XML [1] tem sido apreciada por diversos autores em cenários para os quais a linguagem dificilmente foi considerada. Tal deve-se de facto à facilidade de criar linguagens específicas baseadas em XML e às capacidades da tecnologia XML em transformar especificações em outras representações. Como exemplos do que acaba de ser dito, refira-se que o XML tem sido utilizado para descrever comportamentos [2], máquinas de estados finitos [3], arquitecturas do conjunto de instruções em microprocessadores [4], etc.

A integração na tecnologia XML de um motor de transformação (XSLT [5]) fornece uma solução final adequada ao desenvolvimento incremental de especificações e de mecanismos de transformação necessários. Note-se contudo que as linguagens baseadas em XML são, neste contexto, apenas indicadas para representações intermédias, pois a sua legibilidade sofre imenso com o número de elementos no ficheiro XML.

No nosso caso, a tecnologia XML tem sido utilizada como suporte à investigação de técnicas de compilação para arquitecturas específicas [6][7][8] e em ambiente de investigação de novas arquitecturas reconfiguráveis, baseadas em agregados de células em que cada célula é um elemento de processamento [9][10]. Nestes dois casos o XML tem sido utilizado para representar comportamentos e estruturas e tem-

se revelado extremamente importante. Nos ambientes de projecto de arquitecturas específicas e reconfiguráveis, várias ferramentas (compiladores, simuladores, etc.) são utilizadas. O XML tem sido também utilizado como formato intermediário de comunicação entre ferramentas. Como as tecnologias e os padrões da área estão em constante refinamento, o uso de um formato incremental, baseado em XML, garante uma maior independência de tecnologia, capacidade de reutilização e aumenta das possibilidades de expansão e readaptação dos ambientes de projecto.

A investigação de arquitecturas específicas tem o papel fundamental nos actuais e nos futuros sistemas embebidos [11][12]. Tal deve-se em parte à demanda de altos níveis de desempenho e de economia de consumo de energia muitas das vezes apenas compatíveis com soluções especializadas.

Este artigo está organizado da seguinte forma. A próxima secção apresenta a utilização de XML para arquitecturas específicas. A secção 3 ilustra a utilização de XML como suporte de representação das estruturas finais na compilação de Java bytecodes para FPGAs (*Field-Programable Gate Arrays*). Por último, a secção 4 conclui o artigo e traça alguns comentários acerca das potencialidades desta tecnologia.

2. Avaliação antes do Fabrico de Arquitecturas de Agregados com Granulosidade Grossa

As arquitecturas reconfiguráveis, baseadas em agregados de células de granulosidade grossa, têm provado a apetência para aceleração do desempenho de sistemas computacionais [13]. Tem sido dado ultimamente algum relevo a arquitecturas com comportamento *data-driven* [14]. Nestas, as operações são despoletadas aquando da presença de novos dados nos operandos, à semelhança das célebres máquinas *dataflow* [15]. Desta forma, o controlo acaba por ser distribuído e a computação em pipelining natural.

A **Fig. 1**(a) apresenta as entradas e saídas de uma unidade funcional (FU) com comportamento *data-driven* típica. Para além das entradas/saídas usuais existem os sinais que controlam o funcionamento *data-driven*. Estes sinais constituem os sinais necessários para implementar o protocolo de *handshake* (“aperto-de-mão”) utilizado. A **Fig. 1**(b) ilustra a integração da FU numa célula hexagonal da arquitectura e a **Fig. 1**(c) ilustra um exemplo de uma arquitectura hexagonal. Em arquitecturas deste tipo existem várias topologias possíveis e várias formas de comunicação de dados entre as células. São reconhecidas para as diferentes topologias vantagens e desvantagens, mas o impacto destas no desempenho final necessita de esquemas de avaliação rápida, e por isso se revela de extrema importância a investigação e desenvolvimento em ambientes adequados a essa finalidade.

Embora estas arquitecturas tenham vindo a demonstrar algumas vantagens, não têm sido realizados estudos sistemáticos que permitam avaliar certas propriedades antes da opção por um determinado aspecto ao implementar a arquitectura. Neste âmbito temos vindo a desenvolver um ambiente, designado por EDA, que permita o estudo prévio deste género de arquitecturas [9][10]. Na **Fig. 2** é apresentada uma vista geral desse ambiente. Para o desenvolvimento do ambiente temo-nos socorrido da

tecnologia XML de forma a especificar a arquitectura, os parâmetros possíveis de exploração, e os exemplos de teste.

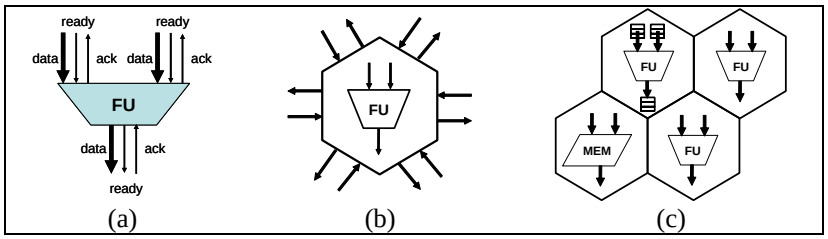


Fig. 1. (a) Unidade funcional *data-driven* (FU) com duas entradas e uma saída; (b) Célula hexagonal com a FU; (c) Agregado hexagonal (FUs podem ter FIFOs nas entradas e nas saídas)

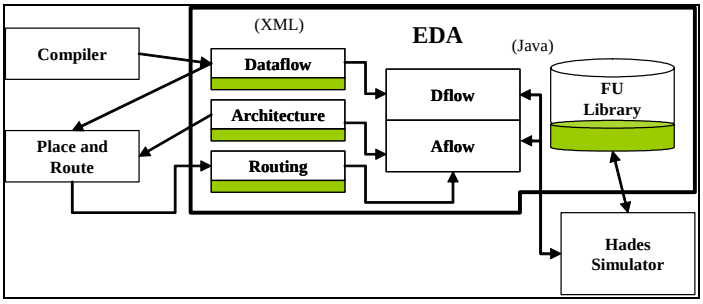


Fig. 2. Ambiente de exploração de arquitecturas do tipo *data-driven* (fonte: [10])

A especificação do *dataflow* é realizada em XML. O XML é também utilizado para especificar a colocação e o encaminhamento dessa especificação na arquitectura sob teste. Cada operador no *dataflow* é directamente implementado pelas unidades funcionais (FUs) existentes nas células da arquitectura. A **Fig. 3** apresenta um exemplo *dataflow*. Na **Fig. 4** pode-se ver a representação em XML do exemplo. A **Fig. 5** ilustra uma arquitectura hexagonal, parte da sua representação XML, o mapeamento do exemplo da **Fig. 3** na arquitectura apresentada e a descrição XML do encaminhamento para esse mapeamento.

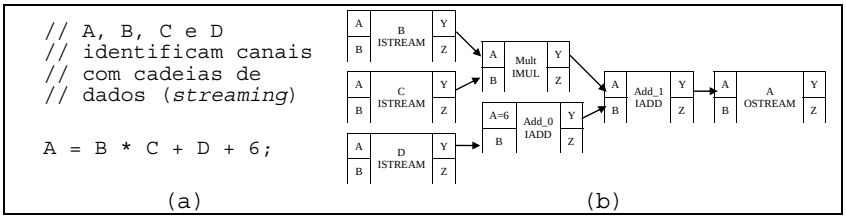


Fig. 3. Exemplo simples: (a) código do exemplo; (b) grafo de uma implementação *dataflow*

Para especificar o comportamento de cada FU, o ambiente utiliza a linguagem Java. A simulação da arquitectura é realizada pela ferramenta HADES [16][17][18] tendo por base uma biblioteca desenvolvida de FUs com comportamento *data-driven*.

```
…
<DESIGN  trace="true">
  <COMPONENT unit="IO" operation="ISTREAM" name="B" file="Data_B.txt"/>
  <COMPONENT unit="IO" operation="ISTREAM" name="C" file="Data_C.txt"/>
  <COMPONENT unit="IO" operation="ISTREAM" name="D" file="Data_D.txt"/>
  <COMPONENT unit="ALU" operation="IMUL" name="Mult"/>
  <COMPONENT unit="ALU" operation="IADD" name="Add_0">
      <PORT name="A" value="6" />
  </COMPONENT>
  <COMPONENT unit="ALU" operation="IADD" name="Add_1"/>
  <COMPONENT unit="IO" operation="OSTREAM" name="A" file="Data_A.txt"/>
  <SIGNAL name="wire1">
      <SOURCE name="B" port="Y"/>
      <SINK name="Mult" port="A"/>
  </SIGNAL>
  …
  <SIGNAL name="wire6">
      <SOURCE name="Add_1" port="Y"/>
      <SINK name="A" port="A"/>
  </SIGNAL>
</DESIGN>
```

Fig. 4. Exemplo simples: parte da representação XML do grafo da **Fig. 3**(b)

Como se pode ver pela **Fig. 5**(b), a definição da arquitectura utiliza regras subjacentes a elementos XML que permitem especificar agregados de forma simplificada. O elemento:

```
<ARRAY type="HEXA">
```

utiliza a propriedade "HEXA" que define a topologia da arquitectura. As topologias aceites até ao momento englobam também a "OCTAL" e a "QUADRANGULAR".

Existem regras que definem vários elementos da arquitectura homogéneos:

```
<LAYOUT length="3" width="3" symbol="I"/>
```

que define uma arquitectura de 3x3 elementos do símbolo "I" (este símbolo havia sido definido em linhas de XML anteriores como representante de células de I/O, ver **Fig. 5**(b)).

As duas linhas seguintes:

```
<LAYOUT x="2" y="0" symbol="M"/>

<LAYOUT x="2" y="2" symbol="M"/>
```

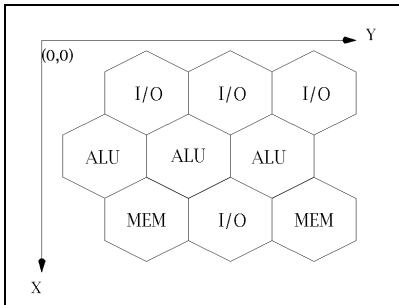
redefinem as células (2, 0) e (2, 2) como sendo do símbolo "M" que anteriormente havia sido definido como correspondente às células do tipo MEM (ver **Fig. 5**(b)).

O elemento final:

```
<LAYOUT x="1" symbol="A"/>
```

indica que todas as células da linha 1 são do tipo "A", previamente definido como ALU. Desta forma obtém-se a arquitectura hexagonal representada na **Fig. 5**(a). A definição da arquitectura também inclui as propriedades referentes ao tipo de

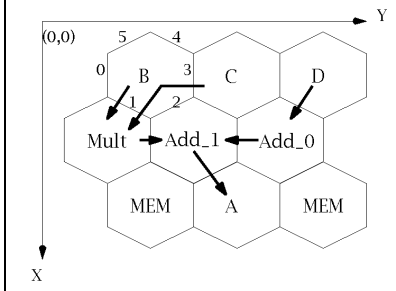
comunicação entre células da arquitectura (de entrada, de saída, bidireccional, etc.) e ao número de interligações. É considerado que os recursos de comunicação entre células são semelhantes para todos os lados de cada célula da arquitectura.



(a)

```
<?xml version="1.0" encoding="UTF-8"?>
<ARRAY type=”HEXA”>
   <CELL name="ALU" symbol="A"/>
   <CELL name="IO" symbol="I"/>
   <CELL name="MEM" symbol="M"/>
   <!-- box 3x3 with same type of cells -->
   <LAYOUT length="3" width="3" symbol="I"/>
   <!-- override special cells -->
   <LAYOUT x="2" y="0" symbol="M"/>
   <LAYOUT x="2" y="2" symbol="M"/>
   <!-- all the cells in line 1 are of type ALU-->
   <LAYOUT x="1" symbol="A"/>
</ARRAY>
```

(b)



(c)

```
?xml version="1.0" encoding="UTF-8"?>
<ROUTING>
   <ROUTE x="0" y="0">
        <CONNECT port="Y" border="1"/>
        <CONNECT border="3" border="1"/>
   </ROUTE>
   <ROUTE x="0" y="1">
         <CONNECT port="Y" border="0"/>
   </ROUTE>
   …
   <ROUTE x="1" y="0">
         <CONNECT border="4" port="A"/>
         <CONNECT border="4" port="B"/>
         <CONNECT port="y" border="3"/>
   </ROUTE>
   …
   <ROUTE x="2" y="1">
         <CONNECT border="5" port="A"/>
   </ROUTE>
</ROUTING>
```

(d)

Fig. 5. (a) Arquitectura hexagonal simples; (b) parte do ficheiro XML que especifica a arquitectura; (c) exemplo *dataflow*; (d) descrição XML que especifica o encaminhamento do exemplo *dataflow* na arquitectura hexagonal anterior

Mediante a utilização de transformadores XSL, as especificações descritas anteriormente são transformadas em código Java responsável pela modelação da arquitectura a simular utilizando o HADES. A programação da arquitectura, tendo por base um determinado exemplo e a informação relativa à colocação e encaminhamentos necessários para interligação das unidades funcionais, descritos em XML, é realizada por código Java especifico que é gerado por transformadores XSLT. O código Java gerado tem por base uma biblioteca de classes Java relacionadas com elementos da arquitectura.

3. Arquitecturas Específicas para FPGAs

A compilação de programas em linguagens de software, como por exemplo de Java [19], para arquitecturas computacionais reconfiguráveis é um tópico de investigação extremamente importante. Tal deve-se ao facto de haver indícios claros de que os sistemas embebidos do futuro utilizarão arquitecturas computacionais mais variadas do que os microprocessadores tradicionais baseados no modelo de von-Neumann. Tendo os FPGAs se tornado em verdadeiras soluções plataforma – poderemos ter num único integrado sistemas completos que podem incluir um ou mais microprocessadores e arquitecturas específicas – é provável que o componente principal de sistemas embebidos complexos possa ser baseado em FPGAs. Neste caso, a geração de uma arquitectura específica que esteja adaptada ao problema que pretende resolver é uma etapa importante no desenvolvimento deste tipo de sistemas. Contudo, o projecto de arquitecturas específicas para este tipo de dispositivos é bastante complicado e necessita de conhecimentos profundos e de experiência consolidada no projecto de sistemas digitais para estes dispositivos. Neste momento não é possível a um programador tirar partido do FPGA sem essa experiência. É por estas razões que a compilação de algoritmos descritos em linguagem alto-nível é fundamental.

Um compilador desse tipo gera uma arquitectura específica a partir do algoritmo descrito. Essa arquitectura é depois mapeada nos recursos existentes do FPGA. O mapeamento é normalmente efectuado por ferramentas disponibilizadas pelos fabricantes deste tipo de dispositivos.

A arquitectura gerada pelo compilador é constituída por um *datapath* (unidade de dados) e por uma unidade de controlo (constituída por uma ou mais FSMs). Estas unidades são representadas internamente sob a forma de grafos que mais tarde são passados às ferramentas de mapeamento utilizando descrições aceites por estas. Normalmente são utilizadas linguagens de descrição de hardware (VHDL, por exemplo) das quais são depois obtidos os circuitos finais utilizando ferramentas de síntese. É ainda gerado um grafo de transições de reconfiguração (*rtg*), que tem como função especificar o fluxo de configurações de forma a que a execução das partições temporais que definem o mapeamento do algoritmo possa fornecer a funcionalidade inicial. O comportamento representado por este grafo pode depois ser implementado por um microprocessador ou por uma unidade de controlo de reconfigurações.

A representação destas unidades pode ser feita utilizando XML. A **Fig. 6** ilustra os ficheiros XML gerados pelo compilador (a sombreado) e os transformadores utilizados. A utilização de XML nesta fase traduz-se nas seguintes vantagens:

- O compilador fornece uma representação da arquitectura independente das ferramentas de mapeamento utilizadas;
- A utilização de transformadores XSL permite a obtenção de representações finais em vários formatos adequados para a finalidade que se pretende.

O ponto anterior pode ser importante para a utilização de por exemplo ferramentas de mapeamento que aceitem outras linguagens. Neste caso o utilizador poderá utilizar o seu próprio transformador XSL.

No ambiente utilizado estão disponíveis vários transformadores XSL:

- para gerar representações de visualização das estruturas geradas pelo compilador. É utilizada a linguagem *dot* e a ferramenta Graphviz [20];
- para gerar os modelos VHDL das estruturas que podem ser depois sintetizados pelas ferramentas comerciais, como o ISE da Xilinx, por exemplo;
- para gerar representações das estruturas em Java para serem simuladas pelo simulador HADES [12].

A infra-estrutura de teste assenta em dois componentes principais. O primeiro é o ambiente de simulação e edição de sistemas lógicos desenvolvido em Java denominado de HADES. Este simulador recebe como entrada um ficheiro descrevendo a rede do circuito a simular, circuito este que utiliza como elementos lógicos básicos as unidades funcionais implementadas pelo compilador (multiplicadores, somadores, RAMs, etc.). Estes elementos por sua vez encontram-se numa biblioteca de operadores descritos em Java e prontos a serem utilizados pelo HADES.

Por outro lado, todo o processo de teste é automatizado pela aplicação ANT [21] que interpreta um ficheiro XML com a descrição das tarefas a executar. A primeira etapa consiste na tradução destes ficheiros de entrada para um conjunto de representações intermédias. Isto é conseguido mediante a utilização de transformadores XSL. O ficheiro descritor do caminho de dados é então traduzido para o formato interpretado pelo HADES. A máquina de estados é traduzida para Java e é interpretada pelo simulador como sendo mais uma das unidades funcionais presentes na arquitectura. Por sua vez o RTG é também traduzido para Java e irá controlar o fluxo de simulação de cada partição temporal.

As secções seguintes descrevem as unidades principais geradas pelo compilador e as representações XML.

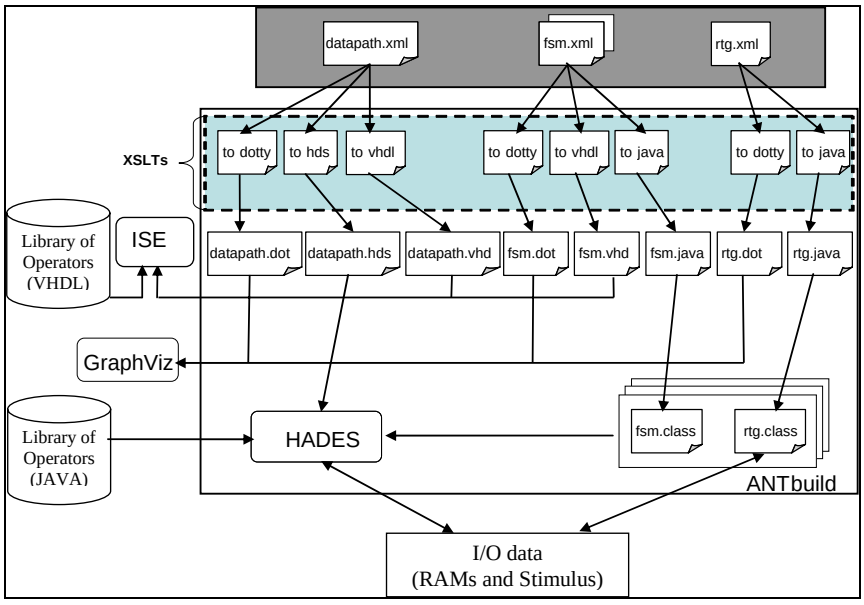


Fig. 6. Utilização da tecnologia XML no *backend* do compilador para arquitecturas específicas

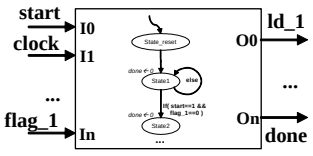
3.1. Especificação da Máquina de Estados

A cada partição temporal corresponde uma unidade de controlo descrita como uma máquina de estados finitos. A **Fig. 7**(a) mostra parte da descrição de uma FSM utilizando o dialecto XML desenvolvido.

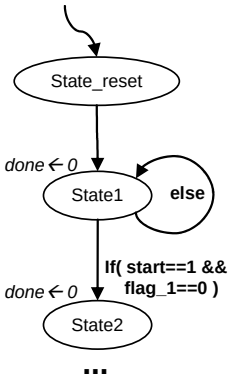
A raiz do dialecto é o elemento FSM. Este é constituido por duas partes distintas definidas por dois elementos filhos, o INTERFACE e o BEHAVIOR, as **Fig. 7**(b) e (c) representam respectivamente esquemas visuais do conteúdo destes elementos no exemplo da **Fig. 7**(a). No INTERFACE são definidos todos os portos de entrada e saída. Isto é feito recorrendo ao elemento PORT, onde são definidos os atributos específicos a cada uma das portas: sentido da ligação (*type*), nome do porto (*name*) e sinal ligado (*signal*). Existe ainda um atributo específico a determinadas portas (*feature*) que indica sinais especiais tais como o sinal de relógio (*clock*) ou o de *reset*.

```
<?xml version="1.0" encoding="UTF-8"?>
<FSM name="fsm0">
  <INTERFACE>
    <PORT type="in" name="I0" signal="start" feature="start"/>
    <PORT type="in" name="I1" signal="clock" feature="clock"/>
    <PORT type="in" name="In" signal="flag_1"/>
    <PORT type="out" name="O0" signal="ld _1"/>
    …
  </INTERFACE>
  <BEHAVIOR>
    <SET signal="done" value="0"/>
    …
    <STATE name="State_reset" type="reset">
      <NEXT name="State1"/>
    </STATE>
    <STATE name="State1">
      <IF>
        <AND>
          <TEST signal="start" operation="=" value="1"/>
          <TEST signal="flag_1" operation="=" value="0"/>
        </AND>
        <NEXT name="State2"/>
      </IF>
      <ELSE>
        <NEXT name="State1"/>
      </ELSE>
    </STATE>
    …
  </BEHAVIOR>
</FSM>
```

(a)



(b)



(c)

Fig. 7. (a) Descrição parcial de uma FSM. (b) Representação das entradas e saídas da FSM. (c) diagrama de estados da FSM

No elemento BEHAVIOR é descrito o comportamento da máquina. A descrição é iniciada com a declaração de elementos SET de atribuição de valores a sinais de saída. Esta atribuição é válida para todos os estados e no caso do sinal ser definido posteriormente, a última atribuição definida prevalece. Para cada estado é definido um

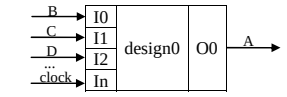
elemento STATE e o estado inicial é sempre definido por aquele que possuir o atributo *type=”reset”*. A transição de estados é conseguida recorrendo ao elemento NEXT, o qual indica o próximo estado da máquina. O comportamento em cada um dos estados pode ser condicional, neste caso é possível a utilização da estrutura de elementos IF/ELSE em que a condição de controlo é definida pelo elemento TEST. Esta condição pode ser composta utilizando os elementos lógicos AND e OR.

3.2 Especificação do caminho de dados

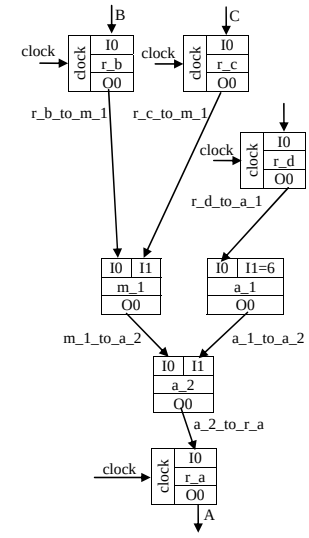
O caminho de dados de cada partição temporal consiste numa estrutura representando as unidades funcionais que o compõem e as respectivas ligações entre si. Na **Fig. 8**(a) está representado parte do *datapath* correspondente ao cálculo da equação da **Fig. 8**(a), utilizando o dialecto XML desenvolvido. O elemento DESIGN é o elemento raiz do dialecto que é constituído por três tipos de elementos filhos. Um elemento INTERFACE que tal como na descrição da FSM descreve os portos de entrada e saída, um ou mais elementos COMPONENT que representam cada uma das unidades funcionais constituintes e um conjunto de elementos SIGNAL que descrevem as ligações entre os vários componentes.

```
<?xml version="1.0" encoding="UTF-8"?>
<DESIGN name = “design0”>
  <INTERFACE>
    <PORT type="in" name="clock" signal="clock" feature="clock"/>
    <PORT type="in" name="B" signal="b" bitwidth="32"/>
    <PORT type="in" name="C" signal="c” bitwidth="32" />
    <PORT type="in" name="D" signal="d” bitwidth="32" />
    <PORT type="out" name="A" signal="a" />
  </INTERFACE>
  <COMPONENT unit="reg_op" name="r_b" bitwidth="32">
    <PORT type="in" signal="clock" name="clock" feature="clock"/>
  </COMPONENT>
  <COMPONENT unit="add_op" name="a_1" bitwidth="32">
    <PORT name="I0" type="in" />
    <PORT name="I1" type="in" value=”6” />
    <PORT name="O0" type="out" />
  </COMPONENT>
  <COMPONENT unit="add_op" name="a_2" bitwidth="32"/>
  …
  <SIGNAL name="r_d_to_a_1" bitwidth="32">
    <SOURCE name="r_d" port="O0"/>
    <SINK name="a_1" port="I0">
      <PROBE/>
    </SINK>
  </SIGNAL>
  <SIGNAL name="a_1_to_a_2" bitwidth="32">
    <SOURCE name="a_1" port="O0"/>
    <SINK name="a_2" port="I2"/>
    </SIGNAL>
  …
</DESIGN>
```

(a)



(b)



(c)

Fig. 8. (a) Descrição parcial do *datapath* correspondente à equação representada na Figura 2(a). (b) Representação do interface de entrada/saída do *datapath*. (c) Diagrama ilustrativo do *datapath*, representando os componentes e sinais de ligação

Cada componente é identificado pela sua unidade funcional (*unit*) e por um nome único (*name*). No caso dos portos de entrada/saída do componente possuírem todos o mesmo tamanho em bits, o elemento *bitwidth* é especificado. Em caso contrário, ou em caso da necessidade de atribuir um valor constante a uma das entradas do componente o elemento PORT terá de ser utilizado para cada um dos seus portos. A atribuição do valor constante ao porto em questão é efectuado utilizando o atributo *value*.

Todos os componentes que necessitem de sinal de relógio terão de ter especificado o porto ao qual o sinal é ligado utilizando o elemento PORT, ver por exemplo o componente de nome *r_b* na **Fig. 8**(a).

As ligações entre portos dos vários componentes são especificadas por elementos SIGNAL. Cada um destes elementos especifica uma ligação, em que é especificado o porto de origem do sinal (SOURCE) e os seus portos de destino (um ou mais elementos SINK). A cada um dos sinais é atribuído o tamanho em bits dos portos de origem e destino.

Para efeitos de teste e despistagem de erros, foi ainda especificado um elemento especial, com a função de indicar quais os sinais que durante a simulação deverão ser escritos para o ficheiro de saída de dados. Este elemento é o PROBE e é especificado como elemento filho do sinal em questão.

4. Conclusões

Este artigo descreve a utilização da tecnologia XML na representação de arquitecturas específicas e de estruturas computacionais. A tecnologia tem permitido os desenvolvimentos incrementais bastante úteis em projectos de investigação, já que novas ideias vão surgindo continuamente e existe a necessidade quase constante de incluir novas propriedades. Embora os transformadores de XSL adicionem atrasos na execução das ferramentas, face por exemplo à geração directa das representações finais, o tempo de execução permanece aceitável.

A linguagem XML tem permitido definir dialectos com semânticas associadas de um modo eficiente e com menores custos de desenvolvimento.

Os transformadores de XSL têm permitido a tradução de uma determinada estrutura ou funcionalidade em representações especificadas na linguagem necessária para a implementação, simulação, visualização, etc.

O trabalho futuro contempla a adição de mais propriedades aos dialectos XML apresentados, de forma a que a exploração de arquitecturas específicas utilizando os trabalhos apresentados neste artigo possa ser ainda mais abrangente e profícua.

Agradecimentos

Os autores agradecem o apoio concedido pela Fundação para a Ciência e Tecnologia (FCT) – programas FEDER e POSI – no âmbito do projecto CHIADO (POSI/CHS/48018/2002).

Referências

1. W3C: Extensible markup language (xml), http://www.w3.org/XML/ (1996-2003).
2. J. E. Coffland, and A. D. Pimentel, "A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems", in *Proc. of the 18th ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA, March 2003, pp. 666-671.
3. Eric Keller, Gordon J. Brebner, Philip James-Roxby, "Software Decelerators", in *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 2003. Peter Cheung, George Constantinides, José T. Sousa (Editors), LNCS 2778, Springer Verlag, pp. 385-395.
4. Shay P. Seng, Krishna V. Palem, Rodric M. Rabbah, Weng-Fai Wong, Wayne Luk and P.Y.K. Cheung, "PD-XML: Extensible Markup Language For Processor Description," In *Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT'02)*, December 2002, pp. 437- 440.
5. W3C: The extensible stylesheet language family (xsl), http://www.w3.org/Style/XSL (1996-2003).
6. João M. P. Cardoso, "CHIADO: compilation of high-level computationally intensive algorithms to dynamically reconfigurable computing systems," in *SPIE Microtechnologies for the New Millennium 2005 Symposium*, Seville, Spain, May 9-11, 2005, SPIE Vol. 5837, pp. 893-901.
7. Rui Rodrigues, and João M. P. Cardoso, "A Test Infrastructure for Compilers Targeting FPGAs," in *International Workshop on Applied Reconfigurable Computing (ARC2005)*, held in conjunction with *IADIS International Conference Applied Computing 2005*, Algarve 22-23, Portugal, IADIS Press, pp. 168-175.
8. Rui Rodrigues, and João M. P. Cardoso, "An Infrastructure to Functionally Test Designs Generated by Compilers Targeting FPGAs," Interactive Presentation at the *Design, Automation and Test in Europe Conference (DATE'05)*, Munich, Germany, March 7-11, 2005, IEEE Computer Society Press, pp. 30-31.
9. Ricardo Ferreira, João M. P. Cardoso, and Horácio C. Neto, "An Environment for Exploring Data-Driven Architectures," in *14th Int'l Conference on Field Programmable Logic and Applications (FPL'04)*, LNCS 3203, Springer-Verlag, 2004, pp. 1022-1026.
10. Ricardo Ferreira, João M. P. Cardoso, Andre Toledo, and Horácio C. Neto, "Data-driven Regular Reconfigurable Arrays: Design Space Exploration and Mapping," in *Embedded Computer Systems: Architectures, Modeling, and Simulation 5th International Workshop*, Timo D. Hämäläinen, Andy D. Pimentel, Jarmo Takala, Stamatis Vassiliadis (Eds.), SAMOS 2005, Samos, Greece, July 18-20, 2005, LNCS 3553 Springer, pp. 41-50.
11. Jürgen Becker, and Reiner Hartenstein, "Configware and morphware going mainstream," in *Journal of Systems Architecture: the EUROMICRO journal*, vol. 49, Issue 4-6, September 2003, pp. 127-142.
12. Francisco Barat, Rudy Lauwereins, and Geert Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective," in *IEEE Transactions on Software Engineering*, vol. 28, no. 9, September 2002, pp. 847-862.
13. R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," In *Int'l Conference on Design, Automation and Test in Europe (DATE'01)*, Munich, Germany, March 12-15, 2001, pp. 642-649.
14. João M. P. Cardoso, "Data-driven array architectures: a rebirth?," in *SPIE Microtechnologies for the New Millennium 2005 Symposium*, Seville, Spain, May 9-11, 2005, SPIE Vol. 5837, pp. 479-490.
15. A. H. Veen, "Dataflow machine architecture," in *ACM Computing Surveys*, Vol. 18, Issue 4, 1986, pp. 365-396.
16. Norman Hendrich, "HADES: The Hamburg Design System," in *ASA'98, European Academic Software Award/Alt-C Conference*, Oxford, 19-21 Sep. 1998.

17. Norman Hendrich, “A Java-based Framework for Simulation and Teaching,” in *3rd European Workshop on Microelectronics Education (EWME’00)*, Aix en Provence, France, 18-19, May 2000, Kluwer Academic Publishers, pp. 285-288.
18. ____, *Hades: Interactive Simulation Framework*, http://tech-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html
19. João M. P. Cardoso, and Horácio C. Neto, “Compilation for FPGA-Based Reconfigurable Hardware,” in *IEEE Design & Test of Computers Magazine*, March/April, 2003, vol. 20, no. 2, pp. 65-75.
20. *Graphviz - open source graph drawing software*: http://www.research.att.com/sw/tools/graphviz/.
21. The apache ant project: http://ant.apache.org/.