MODELAGEM FUNCIONAL E COM PRECISÃO DE CICLOS DO PROCESSADOR NIOS 2

Guilherme Quentel Melo e Luiz C. V. dos Santos

Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
Florianópolis, SC, Brasil

{saddan, santos}@inf.ufsc.br

ABSTRACT

Transaction-level modeling (TLM) is the key description style for SoC design at the Electronic System Level (ESL), since it allows the early development of hardware-dependent software. TLM requires both purely functional and cycle-accurate processor models. This paper describes two TLM-compatible models for the Altera Nios2 processor. The models were experimentally validated with well-known benchmarks. Since Nios 2 is a highly popular processor and since the proposed models are available under GPL license, they are expected to have a huge number of users.

1. INTRODUÇÃO

As nanotecnologias permitiram acomodar sistemas complexos de hardware e software em um único circuito integrado, dando origem aos *Systems-on-Chip* (SoCs). Um SoC contém processadores, memórias, meios de conexão e componentes para entrada /saída e aceleração de tarefas.

A crescente complexidade dos SoCs, o alto custo das máscaras e o custo crescente de engenharia não recorrente deram origem a um paradigma de projeto baseado em plataforma [1]. Uma *plataforma* consiste na adoção de uma arquitetura de referência que atende a um determinado domínio de aplicação e no reuso de blocos de propriedade intelectual (IPs).

Plataformas podem ser descritas em diferentes níveis e estilos de abstração, tais como o *nível de transferência entre registradores* (RTL), o estilo *funcional com precisão de ciclos* (CAM) e o estilo *transacional* (TLM) [2]. Este último baseia-se em descrições funcionais dos IPs e admite duas variantes: uma não temporizada que se limita à *perspectiva do programador* (PV) e outra com temporização aproximada onde se combina a *perspectiva do programador com temporização* (PVT).

A linguagem SystemC [3] mostra-se a melhor alternativa para a modelagem em nível de sistema (ESL) e admite vários níveis e estilos de descrição (RTL, CAM, TLM). Em especial, o estilo TLM foi recentemente padronizado [4] e mostra-se bastante promissor para a diminuição do time-to-market e para o projeto concorrente de hardware e software, conforme relatos da indústria [5].

A modelagem TLM é a tecnologia-chave para o projeto de SoCs em nível de sistema (ESL). Para viabilizá-la, precisa-se de modelos dos vários IPs para compor uma plataforma. Entretanto, a modelagem de processadores diretamente em SystemC não seria pragmática, pois admitiria diversos estilos de descrição, o que dificultaria a geração automática do kit de ferramentas necessário para o desenvolvimento de software embarcado (compilador, montador, ligador, simulador, depurador, etc.). Para a geração automática desse kit, costuma-se utilizar linguagens de descrição de arquiteturas ou *architecture description languages* (ADLs).

A ADL ArchC [6] tem a vantagem de ser distribuída sob licença GPL e gera modelos SystemC compatíveis com TLM. Por um lado, o modelo formal descrito com a ADL permite a geração automática do kit de ferramentas para o desenvolvimento de software. Por outro, ao invés de gerar-se um mero simulador, um modelo de simulação é encapsulado dentro de um módulo SystemC, compatibilizando-o com TLM.

Este trabalho adota a ADL ArchC [6] para a modelagem do processador Nios 2 [7] da Altera, resultando em dois modelos distintos: um puramente funcional (compatível com a variante TLM/PV) e um com precisão de ciclos (compatível com a variante TLM/PV+T). A descrição puramente funcional considera apenas o efeito produzido por cada instrução, abstraindo as informações de temporização. A descrição com precisão de ciclos, modela parte da organização do processador como, por exemplo, estágios de *pipeline*, que tem impacto na temporização das instruções.

Portanto, a contribuição deste trabalho é um par de modelos compatíveis com o paradigma de projeto contemporâneo de SoCs. Os modelos desenvolvidos são

disponibilizados sob licença GPL, juntamente com outros modelos já armazenados em repositório [6].

Este artigo está organizado da seguinte forma: a Seção 2 revisa os trabalhos correlatos; a Seção 3 descreve os modelos puramente funcional e com precisão de ciclos do Nios 2 e algumas de suas características; a Seção 4 apresenta os resultados experimentais, Seção 5 contém conclusões e perspectivas de trabalhos futuros.

2. TRABALHOS CORRELATOS

A ADL adotada para descrição, ArchC, foi desenvolvida pelo *Computer Systems Laboratory* (LSC) da Universidade de Campinas (Unicamp), Brasil.

Juntamente com a ADL, são disponibilizados alguns geradores de ferramentas para desenvolvimento e depuração de código. Dada uma descrição ArchC de um processador, é gerada uma cadeia de ferramentas incluindo montador, ligador, desmontador e depurador. Além disso, a partir da mesma descrição, geradores de simuladores podem criar modelos (puramente funcionais ou com precisão de ciclos) escritos em SystemC para a CPU descrita.

Há vários modelos de processadores já disponíveis no repositório ArchC. Alguns processadores dispõem de modelos funcionais e com precisão de ciclos, tais como PowerPC, MIPS-I, SPARC-V8, Intel 8051 e PIC16F84. Outros possuem apenas o modelo puramente funcional.

À exceção de um protótipo inicial (não certificado) para o processador Nios2 (veja Seção 6), não é do conhecimento dos autores a existência de modelos ArchC para o processador Nios2 em domínio público. Assim, a motivação deste trabalho foi contribuir com dois modelos para um processador de uso bastante difundido em sistemas embarcados, permitindo utilizá-lo em descrições TLM de SoCs.

3. DESCRIÇÃO DOS MODELOS

O Nios 2 [7] é um processador RISC com muitas semelhanças com o processador MIPS [8]. Ele possui 38 registradores de 32 bits, sendo 32 de uso geral (r0 a r31) e 6 de controle. Há 3 formatos de instrução: R, I e J. Eles são idênticos aos formatos utilizados pelo MIPS.

3.1. Descrição do modelo funcional

O modelo funcional é composto de 5 descrições, organizadas em arquivos distintos.

A Figura 1 mostra um fragmento do arquivo que descreve os principais parâmetros do processador (niosIIf.ac). Nela pode-se observar características do processador, tais como, o tamanho da palavra (32 bits), o banco contendo 32 registradores, os registradores de status e de controle e uma memória de 5 MB. Ao final, é especificado o arquivo que contém a descrição da arquitetura do conjunto de instruções (niosIIf-isa.ac) e definido o *endian* do processador (little endian).

```
AC_ARCH(niosIIf)
{
  ac_wordsize 32;
  ac_regbank RB:32;

  ac_reg status_reg_pie;
  ac_reg status_reg_user;

  ac_reg estatus_reg_pie;
  ac_reg estatus_reg_user;

  ac_reg bstatus_reg_pie;
  ac_reg bstatus_reg_user;

  ac_reg ienable_reg;
  ac_reg ipending_reg;

  ac_mem DC:5M;

  ARCH_CTOR(niosIIf)
  {
    ac_isa("niosIIf-isa.ac");
    set_endian("little");
  };
};
```

Figura 1 – Descrição dos parâmetros do processador

A Figura 2 ilustra apenas um fragmento da descrição do conjunto de instruções do modelo (arquivo niosIIf-isa.ac), pois o arquivo completo é bastante extenso. Note que a descrição contém informações como formato, mnemônico e tamanho de cada instrução, bem como informações necessárias para decodificação.

Primeiramente, pode-se observar a declaração dos formatos do Nios 2. Por exemplo, o formato J é declarado contendo os campos *op* e *imm26*, de 6 e 26 bits respectivamente.

Mais adiante, são declaradas as instruções com seus devidos tipos. Em ac_asm_map é feito um mapeamento entre nomes de registradores e seus valores. Em ISA_CTOR, as instruções são associadas a seus mnemônicos em *assembly* e seus códigos para decodificação. Por último, são declaradas as pseudoinstruções.

A Figura 3 ilustra um fragmento da descrição de comportamentos das instruções (arquivo niosIIf-isa.cpp). O fragmento mostra a implementação de uma instrução *add* no modelo funcional: escreve-se no registrador destino o resultado da soma dos registradores-fonte.

Além dos três arquivos exemplificados nas figuras, há também um arquivo de descrição de chamadas de sistema (niosIIf_syscall.cpp) e um arquivo de suporte à depuração (niosIIf_gdb_funcs.cpp). O primeiro informa, por exemplo, como se retorna de uma chamada de sistema e onde se localizam os argumentos de um programa. O segundo mostra como ler e escrever o conteúdo de um registrador.

Embora, estes não sejam arquivos obrigatórios de uma descrição em ArchC, eles contém informações que possibilitam a emulação de chamadas de sistema operacional na máquina hospedeira e a depuração de programas, usando o depurador GNU GDB.

```
AC_ISA(niosII)
{
  ac_format Type_I = "%op:6 %imm16:16:s %rB:5 %rA:5";
  ac_format Type_R = "%op:6 %imm5:5 %func:6 %rC:5
%rB:5 %rA:5";
  ac_format Type_J = "%op:6 %imm26:26";

  ac_instr<Type_I> addi, andhi, andi, beq;
  ac_instr<Type_R> add, And, Break, bret, callr;
  ac_instr<Type_J> call;

  ac_asm_map reg{
        "$"[0..31] = [0..31];
        "r"[0..31] = [0..31];
        "$zero" = 0;
        "zero" = 0;
        "at" = 1;
        "et" = 24;
        "bt" = 25;
        "gp" = 26;
        "sp" = 27;
        "fp" = 28;
  }
  ISA_CTOR(niosII)
  {
    add.set_asm("add %reg, %reg, %reg", rC, rA, rB);
    add.set_decoder(op=0x3a, imm5=0, func=0x31);

    pseudo_instr("subi %reg, %reg, %imm"){
        "addi %0, %1,- %2";
    }
  };
};
```

Figura 2 – Descrição do conjunto de instruções

```
//!Instruction add behavior method.
void ac_behavior( add )
{
  RB.write(rC, RB.read(rA) + RB.read(rB));
}
```

Figura 3 – Descrição de comportamento de instrução

3.2. Descrição do modelo com precisão de ciclos

O modelo com precisão de ciclos é uma extensão do modelo puramente funcional, onde a temporização resulta da inclusão da estrutura dos estágios de *pipeline*. Duas descrições precisam ser estendidas: a descrição dos parâmetros da arquitetura e a descrição dos comportamentos.

A Figura 4 mostra as principais modificações na descrição de parâmetros da arquitetura (niosIIf.ac). Observe a declaração dos formatos dos registradores do *pipeline*, a declaração dos próprios registradores que isolam os estágios do *pipeline* e a declaração dos estágios propriamente ditos.

```
// Pipeline register formats.
ac_format Fmt_F_D = "%npc:32";
ac_format Fmt_D_E = "%npc:32 %data1:32 %data2:32
    %imm:16:s %imm5:5 rA:5 %rB:5 %rC:5 %regwrite:1
    %memread:1 %memwrite:1";
ac_format ... = ...
   ...
// Pipeline registers.
ac_reg<Fmt_F_D> F_D;
ac_reg<Fmt_D_E> D_E;
ac_reg<Fmt_E_M> E_M;
ac_reg<Fmt_M_A> M_A;
ac_reg<Fmt_A_W> A_W;

// Pipeline stages. (Fetch, Decode, Execute,
// Memory, Align, Writeback)
ac_pipe pipe = {F, D, E, M, A, W};
```

Figura 4 – Extensão da descrição de parâmetros

```
void ac_behavior( add ){
  switch( stage ) {
  case F:
    break;
  case D:
    break;
  case E:
    E_M.alures = ex_value1 + ex_value2;
    E_M.rdest = D_E.rC;
    break;
  case M:
    break;
  case A:
    break;
  case W:
    break;
  default:
    break;
  }
};
```

Figura 5 – Extensão dos comportamentos

A Figura 5 mostra as extensões em um fragmento da descrição de comportamentos para a instrução *add*. Essa descrição torna explícitos todos os eventos que ocorrem dentro de cada estágio do *pipeline*. Quando algum dado de um estágio precisa ser passado para outro, escreve-se nos campos dos registradores de *pipeline*.

4. RESULTADOS EXPERIMENTAIS

Os experimentos foram executados em um processador Pentium 4 (3,0 GHz; 1GB de memória principal). Utilizou-se o ArchC 1.6.0 e o compilador GCC 3.4.1. Os modelos foram validados com os conjuntos de *benchmarks Mediabench* [9] e *Mibench* [10].

A Tabela 1 mostra os resultados obtidos com os programas do *benchmark Mibench*.

Tabela 1 – Resultados do *benchmark Mibench*

Nome do programa	Número de instruções	Instruções executadas	Tempo (s) simulação
basicmath-small	15191	1.561.063.336	215,0
basicmath-large	15369	22.039.599.769	3767,0
bitcount-small	10763	41.479.000	5,6
bitcount-large	10763	622.394.114	83,0
qsort-small	13745	16.136.424	2,6
qsort-large	15995	192.013.514	31,0
susan-small corners	18912	3.759.468	0,7
susan-small edges	18912	6.880.985	1,3
susan-small smoothing	18912	31.118.438	4,9
susan-large corners	18912	46.385.816	9,2
susan-large edges	18912	163.131.547	30,5
susan-large smoothing	18912	332.160.363	48,0
dijkstra-small	13567	48.535.523	7,1
dijkstra-large	13567	204.657.540	29,4
patricia-small	14394	309.980.718	35,0
patricia-large	14394	1.964.724.183	270,0
adpcm-small encoder	9747	29.179.525	2,3
adpcm-small decoder	9741	26.270.157	2,1
adpcm-large encoder	9747	579.119.635	49,2
adpcm-large decoder	9741	518.201.806	44,6
crc32-small	10369	38.469.170	2,8
crc32-large	10369	747.721.150	62,3
fft-small	13280	898.972.685	112,0
fft-small inv	13280	2.174.894.392	312,0
gsm-small encode	19847	25.404.166	4,3
gsm-small decode	19847	14.326.004	2,3
gsm-large encode	19847	1.369.426.957	231,0
gsm-large decode	19847	779.962.071	123,0

Os resultados obtidos com o modelo funcional coincidiram com os esperados para todos os casos testados. O modelo foi enviado à equipe ArchC pra certificação e disponibilização em repositório [6].

A validação do modelo com precisão de ciclos está em andamento.

O modelo funcional foi portado para a versão 2.0 do ArchC. Esse porte permitiu testar as novas ferramentas de geração de depuradores e desmontadores, disponíveis na nova versão. Esse modelo contribuiu para a correção de alguns *bugs* nessas ferramentas, visto que é o primeiro modelo *little endian* disponível para essa ADL.

5. CONCLUSÕES E TRABALHOS FUTUROS

Os modelos propostos têm um grande número de usuários potenciais por serem disponibilizados sob licença GPL e por representarem um processador bastante popular. A correção do modelo funcional é amparada pelo grande número de programas executados corretamente. Estima-se que a validação do modelo com precisão de estará terminada até o final de 2006. Além disso, há ainda umas poucas instruções não implementadas nos dois modelos.

6. AGRADECIMENTOS

Agradecemos a Richard Maciel, autor de um protótipo inicial usado como ponto de partida para produzir o modelo puramente funcional aqui descrito.

REFERÊNCIAS

[1] Sangiovanni-Vincentelli, A. and Martin,. G., "Platform-based design and Software Design and software design methodology for embedded systems", *IEEE Design and Test*, 18(6): 23-33, 2001.

[2] Maillet-Contoz, L. and Ghenassia, F., "Transaction Level Modeling: An Abstraction Beyond RTL". In Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer, 2005, chapter 2.

[3] The Open SystemC Initiative. URL: http://www.systemc.org.

[4] OSCI Standard for SystemC TLM. URL: http://www.systemc.org.

[5] Ghenassia, F. and Clouard, A., "TLM: An Overview and Brief History". In Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer, 2005, chapter 1.

[6] The ArchC Architectural Description Language. URL: http://www.archc.org.

[7] Nios 2 Processor Reference Handbook, 2003. URL: http://www.altera.com/literature.

[8] Patterson, D., Hennessy, J., *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed. Morgan Kaufmann Publishers, 2004.

[9] Mediabench Consortium. Disponível em http://euler.slu.edu/~fritts/mediabench/.

[10] Guthaus, M., et al. "MiBench: A free, commercially representative embedded benchmark suite." *IEEE 4th Annual Workshop on Workload Characterization*. URL: http://www.eecs.umich.edu/mibench/