VII IBERCHIP WorkShop
Montevidéu – Uruguai
Março - 2001

UM AMBIENTE DE COMPILAÇÃO E SIMULAÇÃO PARA PROCESSADORES EMBARCADOS PARAMETRIZÁVEIS

Fernando Gehm Moraes [1]
Ney Laert Vilar Calazans [2]
César Augusto Missio Marcon [3]
Aline Vieira de Mello [4]

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Faculdade de Informática
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4
Telefone: +55 51 320-3611 - Fax: +55 51 320-3621
CEP 90619-900 - Porto Alegre - RS - BRASIL

Abstract

This paper presents an environment to compile and simulate a given processor architecture, used in embedded applications. The designer can configure the processor instruction set and internal organization. The instruction set parameterization allows to fit the instruction set to the user application, for example, some arithmetic operations, as multiplication or division, can improve performance or reduce the cost of the processor to be implemented. The internal organization, composed by a data-path and a control unit reflects the instruction set choices, as the number of internal registers, the arithmetic and logic unit functions and the stack structure. The availability of a flexible, low cost, high performance processor is crucial in system-on-a-chip designs. This environment to compile and simulate a processor architecture includes 5 tools: architecture customizer, assembler, simulator, C compiler and VHDL code generator. This paper presents the first 3 tools, being the C compiler and VHDL code generator under development.

Resumo

Este artigo apresenta um conjunto de ferramentas para compilação e simulação de uma arquitetura de processador a ser utilizada em aplicações embarcadas. A arquitetura desenvolvida é parametrizável tanto no conjunto e formato das instruções quanto na organização interna. A parametrização do conjunto de instruções possibilita ao projetista selecionar as instruções que mais se adaptam a sua aplicação. Por exemplo, a existência ou não de determinadas instruções aritméticas, como multiplicação ou divisão, pode aumentar o desempenho ou reduzir o custo do processador a ser implementado. A parametrização da organização interna do processador reflete as escolhas tomadas na seleção das instruções, como número de registradores, operações da unidade lógico-aritmética ou o funcionamento da pilha. A disponibilidade de um processador que seja flexível, apresente baixo custo em termos de área, sem sacrificar o desempenho é fundamental em projetos de sistemas integrados em um único circuito integrado - SOC (*system-on-a-chip*). O ambiente de compilação e simulação é composto por 5 ferramentas: o configurador da arquitetura, o montador, o simulador, o compilador C e o gerador do código VHDL. O artigo apresenta as 3 primeiras ferramentas, estando o compilador C e gerador do código VHDL em implementação.

[1] Doutor em Informática, opção Microeletrônica (LIRMM, França, 1994), Engenheiro Eletrônico (UFRGS, 1987), Professor Adjunto da Faculdade de Informática/PUCRS.
E-mail: moraes@inf.pucrs.br

[2] Doutor em Ciências Aplicadas, opção Microeletrônica (UCL, Bélgica, 1993), Engenheiro Eletrônico (UFRGS, 1985), Professor Titular da Faculdade de Informática/PUCRS.
E-mail: calazans@inf.pucrs.br

[3] Doutorando em Informática (PUCRS, Brasil), Mestre em Ciência da Computação (UFRGS, 1992), Engenheiro Eletrônico (UFRGS, 1989), Professor da Faculdade de Informática/PUCRS.
E-mail: marcon@inf.pucrs.br

[4] Aluna do Bacharelado em Informática da PUCRS
E-mail: alinev@inf.pucrs.br

1 Introdução

A maior parte do desenvolvimento de métodos e ferramentas para o projeto de sistemas digitais das últimas quatro décadas (60 a 90) derivaram de necessidades relacionadas com o projeto de computadores e sistemas periféricos. Contudo, a ênfase de pesquisa tem se deslocado gradativamente para um mercado muito mais vasto, o de sistemas eletrônicos que entram na composição de produtos de uso específico. Exemplos são automóveis, aeronaves, eletrodomésticos e dispositivos de comunicação pessoal tais como telefones celulares e *pagers*. A situação é tal que a renda bruta oriunda da venda de microprocessadores (de uso prioritário em computadores) deve em breve ser superada por aquela derivada da comercialização de microcontroladores, sobretudo em produtos de uso específico. Sistemas embarcados ou sistemas embutidos são para todos os efeitos, sistemas computacionais que executam uma função específica. Eles possuem a mesma estrutura geral de um computador, mas a especificidade de suas tarefas faz com que não sejam nem usados nem percebidos como um computador [1][2]. Exemplos típicos de sistemas embarcados são chaves ATM; roteadores de rede IP com exigência de Qualidade de Serviço e estações base para comunicação móvel [3]; sistemas de controle automotivo, tais como: controle de emissão de poluentes e controle de sistemas de frenagem ABS e computadores de bordo em aeronaves.

A crescente complexidade de produtos eletrônicos tecnológicos tem conduzido projetistas e pesquisadores da área de ferramentas de apoio ao projeto a elevar cada vez mais o nível de abstração de tarefas como especificação e validação de sistemas digitais. O objetivo primordial é reduzir, ou pelo menos tornar gerenciável a complexidade do projeto de tais produtos com o conseqüente encurtamento do tempo necessário para a chegada do produto ao mercado (em inglês, *time-to-market*), enquanto ao mesmo tempo se procura reduzir o custo do produto (através da redução dos custos de engenharia não-recorrentes) e elevar seu desempenho (pelo alcance de pontos ótimos no espaço de soluções do problema) [4].

O mercado força a redução de custos e tempo de projeto, o que leva os projetistas a desejar deslocar a maior parte da funcionalidade dos sistemas embarcados para o *software*, deixando os elementos de *hardware* dedicados apenas para funcionalidades que necessitam de alto desempenho [4]. Este fato mostra a necessidade de direcionar esforços para analisar e conceber automaticamente a distribuição adequada da funcionalidade do sistema entre o *software* e o *hardware*. De Micheli expõe em [1] que a maioria dos sistemas digitais modernos são programáveis, consistindo de componentes de *software* e *hardware*. Utiliza-se neste artigo o termo "sistemas computacionais" para referir-se a este tipo de sistemas. O mesmo autor define projeto integrado de *hardware* e *software* (*hardware/software codesign*, ou apenas *codesign*) como: "*a busca do alcance dos objetivos em nível de sistema do produto pela exploração da sinergia entre hardware e software, através do projeto concorrente destas entidades*".

As tarefas de projeto de sistemas computacionais embarcados no nível sistêmico podem ser agrupadas em quatro classes principais, que são modelagem, particionamento *hardware*/*software*, síntese e validação:

- A modelagem sistêmica emprega um formalismo de descrição que define a funcionalidade do sistema computacional e um modelo formal subjacente para o qual a descrição inicial deve ser traduzida e do qual são extraídas as informações para as tarefas posteriores de *codesign*. O formalismo de descrição pode ser uma linguagem de programação (e.g.: C, Java, Occam), uma linguagem de descrição de *hardware* (e.g.: Verilog ou VHDL), ou outra linguagem derivada ou não das anteriores (e.g.: HardwareC [5], Cx [6], ESTEREL [7], SDL [3] e Unity [8]);
- O particionamento da especificação em componentes de *hardware*, *software* e interfaces para que os componentes possam se comunicar é talvez a tarefa mais fundamental e complexa do nível sistêmico, a obtenção de área e desempenho dependem fortemente desta tarefa, que raramente é desenvolvida de forma completamente automática;
- A validação do projeto de sistemas embarcados consiste em fornecer garantias de funcionamento integrado correto para as partes de *software* e *hardware* do sistema. A tarefa de validação pode ser realizada através de verificação formal e/ou simulação dos componentes de *hardware* e *software*, de forma isolada (apenas um dos componentes) ou integrada (interação entre o *hardware* e o *software*);
- A síntese engloba as tarefas de construção de modelos de níveis de abstração inferiores ao sistêmico e pode ser dividida em duas classes de tarefas: a síntese do *hardware* e a síntese do *software*. Além do mais, a comunicação entre a implementação de *software* e a implementação de *hardware* implica na síntese de interfaces, tanto no lado do *hardware*, quanto no lado do *software*.

No processo de *codesign*, o projetista parte da especificação de um sistema, analisando o conjunto de requisitos e restrições para a obtenção de uma ou mais descrições abstratas que representem o sistema computacional. Obtidas as descrições, devem ser extraídas informações destas que permitam direcionar a implementação do mesmo para

particionar o sistema em componentes de *hardware* e *software*. Estas informações geralmente estão contidas no conjunto de requisitos e nas restrições do projeto. O particionamento gera novas descrições, tanto para os componentes de *hardware*, quanto para os componentes de *software*. Ambas descrições passam pelo processo de síntese, que gera três elementos distintos: o *software*, o *hardware* e a interface de comunicação entre o *hardware* e o *software*. Para todas as etapas do projeto podem existir estágios de validação, por verificação formal ou simulação, que permitem avaliar o quão correta está a descrição obtida.

De acordo com vários autores [9][10], em menos de 7 anos já existirão no mercado circuitos integrados compostos por mais de um bilhão de transistores. Com esta capacidade de integração, pode-se imaginar a inclusão de um sistema computacional completo em um único *chip*, o que cria o conceito de SOC (*System On a Chip*). Este acréscimo tecnológico certamente força o estudo de novas metodologias de projeto de sistemas. A diferença é que a execução do *software*, geralmente associada a um processador de uso genérico mais elementos de memorização, deverá ser efetuada em um módulo processador dentro do CI, denominado *core processor*, e a execução do *hardware* pode ser obtida com o mapeamento de um circuito dedicado também interno ao CI. Para o processo de *codesign*, o grande ganho desta abordagem está na interface de comunicação, pois nos sistemas atuais o maior gargalo é a perda de desempenho causada pela troca de informações entre o *hardware* e o *software* executados em CIs distintos. Caso os componentes de *hardware* e *software* estejam integrados em um único CI, o desempenho global do sistema tende a ser muito maior. Além do mais, a possibilidade de realizar um sistema completo em um único CI (SOC) pode reduzir o *time-to-market* e criar novas relações de desempenho entre o *hardware* e o *software*.

Até bem pouco tempo, as tarefas de *codesign* voltavam-se sobretudo para a realização de produtos compostos por circuitos comerciais de prateleira (em inglês *off-the-shelf*) e eventualmente circuitos fabricados sob encomenda, tais como ASICs (*Application Specific Integrated Circuits*) ou ASIPs (*Application Specific Instruction Set Processors*) [1]. O grande crescimento da tecnologia de dispositivos eletrônicos configuráveis e reconfiguráveis no campo, aumenta a flexibilidade das implementações e torna mais tênue o limite do que é *hardware* e o que é *software*. Isto ocorre porque estes dispositivos podem ter a funcionalidade de seu *hardware* definida de forma dinâmica pelo usuário. O melhor exemplo de tais dispositivos são os FPGAs baseados em RAM. Tipicamente, a cada configuração distinta (em geral, da RAM de controle), corresponde uma funcionalidade distinta do dispositivo. FPGAs no estado da arte possuem capacidade para implementar circuitos com mais de 3 milhões de portas lógicas equivalentes. A capacidade de FPGAs tem crescido a uma taxa superior à prevista pela Lei de Moore [11]. Este ano, os FPGAs ultrapassarão 10 milhões de portas lógicas equivalentes de capacidade, e poderão conter processadores de uso genérico e memória integrados. Um exemplo é o FPGA Excalibur da Altera [12], que devido a sua arquitetura com memória e mais o *core* de um processador de uso genérico (ARM, MIPS ou NIOS) se torna muito atrativo para desenvolvimento de trabalhos de *codesign*.

Este artigo tem o objetivo de apresentar uma arquitetura configurável para processadores embarcados, implementados em FPGA. A arquitetura é parcialmente parametrizável em termos de módulos funcionais (pilha, número de registradores, operações da ULA) e conjunto de instruções. Com o processador definido, este passa a ser utilizado como um *core processor* que implementa as funções do *software*, enquanto que as demais partes do FPGA são utilizadas para a implementação dos componentes de *hardware* e da interface de comunicação. O grande ganho desta abordagem está em criar um processador dedicado e integrado a uma camada de *hardware*, de forma que parte do sistema possa ser implementado em *hardware* e parte em *software*, e a parte implementada em *software* tenha maior desempenho devido à flexibilidade de configuração do processador embarcado.

Uma possível deficiência está associada a limitações dos FPGAs atuais, os quais dispõem de uma pequena quantidade de memória interna, de forma que sistemas de média e grande complexidade devem ser implementados com uma memória externa, perdendo assim perdendo as vantagens inerentes de SOCs.

Este artigo está organizado da seguinte forma. A seção 2 apresenta a arquitetura básica do processador, destacando a organização do bloco de dados e do conjunto básico de instruções. A seção 3 apresenta o ambiente de desenvolvimento proposto para a geração de processador embarcado. As Seções 4 e 5 descrevem a ferramenta que permite configurar o conjunto de instruções da arquitetura conforme a especificação do usuário e o ambiente de montagem e simulação, respectivamente. Finalmente, são apresentadas algumas conclusões e trabalhos futuros.

2 Descrição da arquitetura

O processador denominado ***R6***, é uma organização Von Neumann (memória de dados/instruções unificada), *Load/Store*, com CPI igual a 2 [13], barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características gerais de máquina RISC, tal como *pipeline* e módulos de

entradas/saída, como tratamento de interrupções. Estas deficiências devem-se ao fato desta arquitetura ter sido originalmente desenvolvida visando o ensino de Organização de Computadores [14], na graduação. Sendo a inclusão de *pipeline* um dos tópicos de Arquitetura I, trabalho da disciplina subsequente. No futuro, pretende-se inserir a evolução da arquitetura R6 com a inclusão de módulos de entrada/saída ao escopo de uma terceira disciplina de graduação (Arquitetura II).

Processadores embarcados comerciais simples, tais como o NIOS [12] da Altera, possuem estrutura semelhante ao R6, diferenciando-se principalmente por já disporem de estruturas de entrada/saída, tais como tratamento de interrupções e *timers*.

A comunicação entre o processador e o meio externo compreende: sinais de temporização, *clock* e *reset*; sinais de acesso à memória, *ce*, *rw*, *address*, *datain* e *dataout*; e sinal que indica término de execução do programa, *e_halt*.

2.1 Implementação da Arquitetura

A implementação da arquitetura feita em VHDL [16], compreende a construção dos dois blocos principais: o bloco de controle e o bloco de dados. Estes blocos são unidos hierarquicamente em um terceiro bloco, constituindo assim o processador. Ambos os blocos são parametrizáveis, seja pelo conjunto de instruções no bloco de controle, seja pelos recursos (registradores, instruções da Unidade Lógica Aritmética, etc.). Isto confere à arquitetura R6 uma boa dose de flexibilidade.

Bloco de Controle

O BC tem por função gerar os comandos para a busca da instrução (*fetch*) e depois enviar os comandos ao bloco de dados para que a instrução seja executada. Sua implementação é uma grande estrutura de seleção que avalia o registrador de instrução corrente e gera as microinstruções necessárias para que o bloco de dados realize as operações sobre dados. O BC é formado por dois processos:

- O primeiro é responsável por decodificar a instrução *halt* e gerar o sinal externo *e_halt,* o qual indica que o processador suspendeu a execução de instruções;
- O segundo processo, mostrado na Figura 1, determina o funcionamento normal do BC. Após o *reset*, na primeira borda de subida do clock, é lançada a microinstrução responsável pela busca de instrução (*fetch*). Na segunda borda do clock, a instrução que foi buscada é avaliada por uma grande estrutura de seleção, escolhendo-se assim a microinstrução correspondente à instrução especificada. Uma vez terminada a execução, inicia-se novo ciclo de busca.

```
process
begin

   wait until rst='0' and ck'event and ck='1';

   -- geração da microinstrução para o FETCH
   uins <= (c1, c0,c0, c1, c0,c0,c0,c0, c0,c0, c1,c0,c0,c0, c1,c1, incB );

   wait until ck'event and ck='1';

   if hlt = '1' then      --  HLT
       uins <= (c0, c0, c0, c0, c0,c0,c0,c0, c0,c0, c0,c0,c0,c0, c0,c0, passaA );
   elsif  op1=x"0"   then
       uins <= (c0, c0,c1, c0, c0,c0,c0,c0, c1,c1, c1,c0,c0,c0, c0,c0, add );   -- execução da soma
   elsif ................
   else
       uins <= (c0, c0, c0, c0, c0,c0,c0,c0, c0,c0, c0,c0,c0,c0, c0,c0, passaA ); -- execução do NOP
   end if;

 end process;
```

Figura 1 - Estrutura de seleção da microinstrução para busca e execução de instruções.

Esta solução tem as vantagens de ser simples de implementar e de fácil modificação. Caso queiramos suprimir uma instrução basta remover do segundo processo duas linhas de código VHDL, que representam a avaliação e a geração da instrução corrente. Analogamente, a mesma facilidade ocorre para a inserção de uma nova instrução, desde que haja caminho no bloco de dados.

Esta característica é muito importante no momento da personalização do processador em função da aplicação do usuário (geração do ASIP), pois permite automatizar facilmente a inserção e remoção de novas instruções. A partir do código objeto do usuário e de um banco de dados com todas as instruções disponíveis, gera-se um bloco de controle personalizado.

Bloco de Dados

A arquitetura contém, no bloco de dados, 15 registradores de uso geral (o registrador índice 0 é a constante 0); registradores para armazenamento da instrução corrente (*IR*), endereço da próxima instrução a executar (*PC*) e ponteiro de pilha (*SP*); uma ULA (Unidade Lógica e Aritmética) com 16 operações e 1 qualificador de estado, utilizado para os saltos. A Figura 2 ilustra a organização do bloco de dados deste processador, assim como a interface com a memória externa. Os sinais em itálico, conectados à cada módulo do processador (registradores, ULA, multiplexadores, *tri-states*) representam os comandos provenientes do bloco de controle. Discussão sobre diferentes estratégias de implementação do bloco de dados pode ser encontrada em [16].

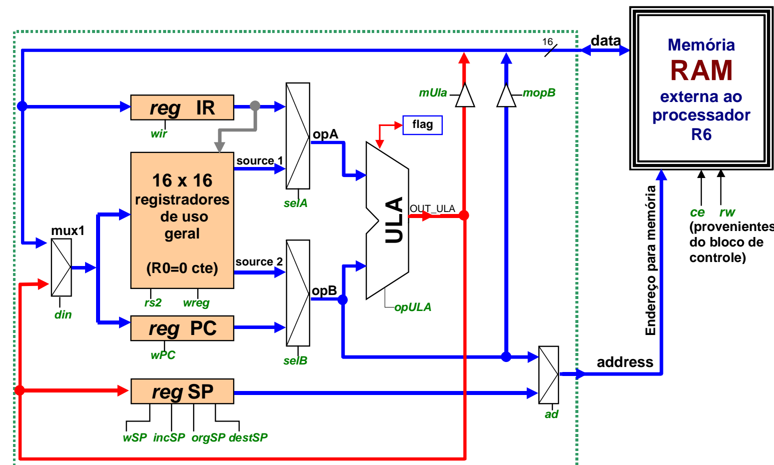

Figura 2 - Bloco de dados da Arquitetura R6, e interface com a memória.

O banco de registradores tem uma porta de acesso para escrita, proveniente do multiplexador *mux1*, e duas portas de leitura, *source 1* e *source 2*. A seleção do registrador a ser alterado ou dos registradores a serem lidos é especificada implicitamente pela instrução corrente, a qual é armazenada no registrador *IR*.

A configuração do processador, em função da aplicação do usuário, deve personalizar (e ordenar) o número de registradores. Por exemplo, caso a aplicação do usuário utilize os registradores R1, R4, R7, R15, o processador gerado deverá conter apenas 4 registradores, renomeando-se os registradores R4, R7 e R15 para R2, R3 e R4. Esta personalização não reduzirá apenas o número de registradores, mas sim todo o hardware de seleção e decodificação, o qual é extremamente custoso em termos de área ocupada no FPGA [16].

A implementação da ULA também deve permitir a personalização do processador, sem sacrifício de área. Uma solução de seleção de operação em função da microinstrução mostrou-se muito custosa em termos de área, pois há uma excessiva replicação de operadores aritméticos. Uma solução que obteve bons resultados em termos de área é a utilização de dois níveis de multiplexação, tal como apresentada na Figura 3. O primeiro nível gera o resultado de uma soma/subtração, e o segundo nível ou seleciona operações simples (lógicas) ou o resultado da soma/subtração, proveniente do primeiro nível.

```
out_ula <=     (BusA or busB)    when uins.ula=ou       else
               (BusA xor busB)   when uins.ula=ouX      else
               ...
                output_adder;

 input_adder1 <=    (not busA)     when uins.ula=negA or uins.ula=notA else
                    (others=>'0') when uins.ula=passaB else
                    ....
                    else busA;
 input_adder2 <=    (not busB)     when uins.ula=AsubB else
                    (others=>'1') when uins.ula=negA or uins.ula=notA or uins.ula=passaA else
                    ....
                    else busB;

 somaAB( input_adder1, input_adder2, cin, output_adder, coutant, cout);
```

Figura 3 – Implementação (parcial) da ULA.

Por fim, o terceiro módulo parametrizável do bloco de dados refere-se às instruções com a pilha. Caso a aplicação do usuário não utilize explicitamente operações de pilha, o registrador SP e toda lógica que o acompanha, pode ser removida.

2.2 Conjunto de Instruções

As instruções possuem formato regular (4 variações do formato básico) e poucos modos de endereçamento. No ciclo de busca de instrução são realizadas duas operações em um único ciclo de relógio: (*i*) leitura da posição de memória apontada por *PC* e (*ii*) incremento do *PC*. As instruções realizadas durante o ciclo de execução podem ser classificadas em 8 grupos, todas executadas em um único ciclo de relógio:

1. Operações **lógico-aritméticas**. Podem envolver 2 registradores fonte (Rs1 e Rs2) e um registrador destino (Rt), realizando-se a função Rt ← Rs1 *opcode* Rs2. *Opcode* assume as funções soma, subtração, e lógico, ou lógico e ou exclusivo. Caso a operação envolva apenas um registrador fonte (Rs), realiza-se a função Rt ← *opcode* Rs. *Opcode* pode especificar as funções de deslocamento à esquerda/direita e inversão lógica (not).
2. Operações de **saltos** e **saltos à subrotinas** com deslocamento curto, relativo a registrador e absoluto. Todos os saltos são condicionais ao qualificador de estado, devendo-se setar este qualificador caso seja necessário um salto incondicional.
3. Operações em **modo imediato curto**. Realizam as seguintes funções: *LDL*, carga da parte baixa de um registrador com constante de 8 bits; *LDH*, carga da parte alta de um registrador com constante de 8 bits; *ADDI* e *SUBI*, soma e subtração com constante de 8 bits (a constante deve ter o sinal estendido).
4. Operações de **carga** de dados, modo indireto (*load)*. Um registrador *Rtarget* recebe o conteúdo de memória apontado pelo registrador *Rsource*.
5. Operações de **escrita** de dados, modo indireto (*store)*. A posição de memória apontada por um *Rtarget* recebe o conteúdo de *Rsouce*.
6. Operações com a **pilha** (além dos saltos já descritos anteriormente): *LDSP*, carga do conteúdo de um registrador no registrador SP; *RTS*, retorno de subrotina; *PUSH* e *POP*, gravam e recuperam conteúdo de registrador na pilha, respectivamente.
7. Operações para manipulação do **qualificador** de estado: *DIF*, *EQU*, *SUP*, *INF*. Fazem com que o qualificador receba '1' se os operandos são diferentes, iguais, se o primeiro for maior que o segundo ou se o primeiro for menor que o segundo, respectivamente. O qualificador recebe '0' caso contrário.
8. Operações **miscelâneas**: *NOP* (nenhuma operação) e *HLT* (suspensão da execução de instruções pelo processador).

2.3 Validação da Organização do Processador

A validação do processador envolve duas etapas. A primeira etapa corresponde à simulação VHDL. O simulador/montador da arquitetura, descrito nas Seções seguintes, gera o código objeto de uma dada aplicação. Este código objeto, por sua vez, é lido pelo simulador VHDL [17], através de comandos pré-estabelecidos em um *test bench* VHDL padrão, o que permite a execução do programa do usuário. O comportamento observado nos diagramas de tempo deve corresponder aos valores obtidos a partir da simulação funcional (Seção 5).

A segunda etapa de validação corresponde à implementação do processador em uma plataforma de prototipação. Em [16] é apresentada a validação de uma versão anterior do processador R6 (o processador *R3*) em uma plataforma de prototipação contendo um FPGA Xilinx 4010 (10000 portas lógicas equivalentes). A implementação em FPGA implicou em desenvolver um controlador de acesso à memória externa. O conjunto processador R3 mais controlador consumiu 365 CLBs (blocos lógicos), dos 400 disponíveis no FPGA (versão com 12 registradores). A atual versão do processador não foi testada em plataforma de prototipação, pois o objetivo agora é testar o ambiente genérico de simulação e compilação. Como a estrutura geral do processador ***R6*** é semelhante ao processador *R3*, a estratégia de implementação é a mesma.

Considerando-se apenas o processador, sem a máquina de estados de controle para acesso à memória, o consumo de área para o processador ***R6*** após a síntese foi de 400 CLBs (100% do FPGA Xilinx 4010), com custo equivalente a 7386 portas lógicas, fornecido pelo relatório de síntese. Isto demonstra que tem-se um processador eficiente em termos de área, que pode ser utilizado como *core processor* em aplicações que necessitem o projeto conjunto de hardware/software.

2.4 Personalização da Arquitetura

A personalização da arquitetura é obtida através da alteração do conjunto de instruções no bloco de controle, do banco de registradores, e de instruções na Unidade Lógico e Aritmética e pilha. Esta etapa, cuja automatização ainda está em estudo, tem uma grande preocupação em evitar o desperdício de área no momento de sintetizar o processador,

gerando-se o *core processor* apenas com os módulos funcionais requeridos pela aplicação do usuário. Desta forma, em projetos conjuntos de *hardware* e *software* disponibilizar-se-á uma maior parcela de área para a parte de *hardware* do projeto, já que o processador está dedicado para a aplicação alvo.

3 Ambiente de desenvolvimento para a geração de processador embarcado

O processador descrito é flexível tanto no conjunto de instruções quanto na sua organização interna. Novas versões podem ser definidas em função de necessidades em termos de instruções ou número de registradores, por exemplo. Visando suprir esta flexibilidade, propõe-se um ambiente de desenvolvimento para a parametrização/geração desta arquitetura. A Figura 4 ilustra o ambiente de desenvolvimento proposto. Este ambiente possui cinco ferramentas: o configurador da arquitetura, o montador, o simulador, o compilador C e o gerador do código VHDL. Destas, o montador e o simulador já estão funcionais, e permitem gerar o código objeto de uma dada aplicação e simular funcionalmente o processador, em função da configuração da arquitetura. O configurador da arquitetura está igualmente funcional.

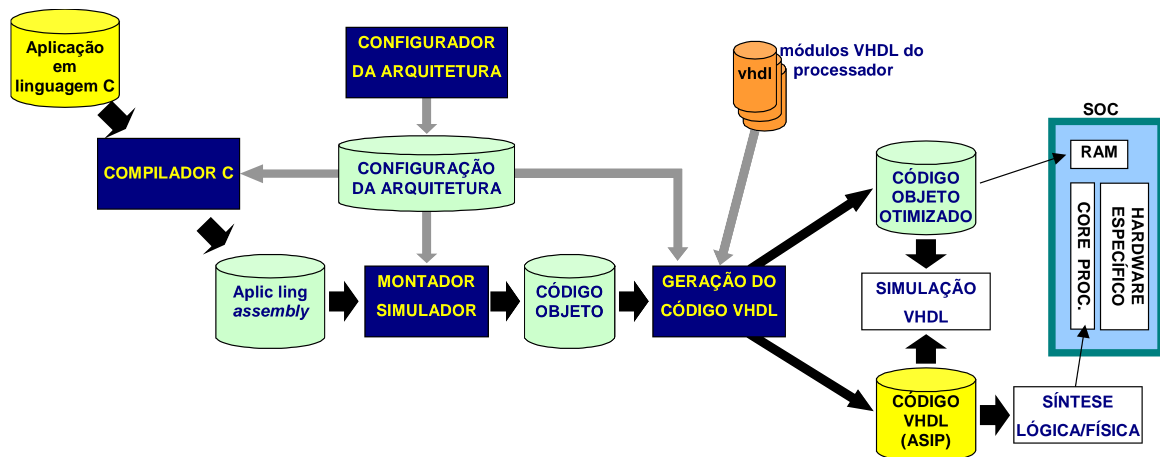

Figura 4 - Ambiente de desenvolvimento para a geração de processador embarcado.

Seja dado um sistema composto de uma parte de *hardware* e de uma parte de *software*. O sistema ideal para o tratamento da parte *software*, seria aquele onde, a partir de uma aplicação em linguagem C pudéssemos obter de forma automatizada o código objeto otimizado para esta aplicação, assim como o ASIP, a ser implementado no circuito integrado. A parte *hardware* seria implementada a partir de, por exemplo VHDL ou Verilog, no restante do circuito.

Para que este sistema opere conforme o esperado, o usuário deve possuir um arquivo de configuração da sua arquitetura. Esta configuração permite ao compilador C gerar a descrição em linguagem de montagem (*assembly*) da parte *software*, a qual é utilizada para a geração do código objeto pelo montador. Este código objeto, a configuração da arquitetura e um banco de dados com módulos VHDL permitem a geração do código objeto otimizado e do código VHDL que representará a descrição personalizado do processador (ASIP). Este ASIP é sintetizado, resultando no *core processor* que executará o código objeto armazenado em memória.

4 Configuração da arquitetura

A ferramenta de configuração, desenvolvida em linguagem Java[1], permite gerar uma descrição para a arquitetura utilizada. A Figura 5-a mostra a janela principal do configurador, onde insere-se um conjunto de parâmetros. Estes parâmetros são gravados em um arquivo de configuração, denominado "*nome_processador*.txt", o qual é utilizado para montagem e simulação de aplicações *assembly*.

O configurador de arquiteturas contém os seguintes parâmetros: (*i*) o nome da arquitetura; (*ii*) o número de registradores; (*iii*) o tamanho da memória de dados e programa; (*iv*) o tamanho da palavra; (*v*) o funcionamento da pilha; (*vi*) os qualificadores de estado; e (*vii*) as instruções da arquitetura. O número de registradores pode variar entre 1 e 16. O tamanho da memória pode ter de 255 a 65535 posições. A atual versão em desenvolvimento tem tamanho

[1] A linguagem Java foi escolhida devido ao fato de ter domínio público, permitindo assim acesso gratuito ao *software*.

fixo de palavra igual a 16 bits. A pilha pode ser pós-incrementada (da posição de memória com endereço menor para a posição de memória com endereço maior) ou pré-decrementada (da posição de memória com endereço maior para a posição de memória com endereço menor).

(a) Definição de parâmetros da arquitetura

(b) Configuração das instruções

Figura 5 - Configurador de Arquiteturas.

Existe nesta ferramenta um conjunto de instruções pré-definidas. O objetivo é associar mnemônicos e campos de operandos a estas instruções pré-definidas, de forma a permitir a leitura do código *assembly* de uma dada aplicação e gerar corretamente o código objeto. É permitido ao usuário definir uma instrução cujo comportamento não esteja definido no configurador. Caso isto ocorra, a montagem pode ser executada, porém a simulação funcional não pode ser feita. Esta nova instrução deverá ser definida na organização do processador, em linguagem VHDL, para que seja possível sintetizá-la.

A Figura 5-b ilustra a janela de configuração de instruções. O primeiro campo, *nome da instrução*, define o mnemônico associado ao código *assembly* correspondente ao comando selecionado (no exemplo, a função ADD).

As instruções podem ocupar 1 ou 2 palavras de memória, devendo-se informar a função de cada campo da instrução. Ao selecionar uma instrução são exibidos os campos obrigatórios (no exemplo Op1, Rt, Rs1, Rs2). A função do usuário é de definir a ordem destes campos na palavra de instrução. Notar que é função do usuário definir o número da instrução (*opcode*) quando o campo for do tipo *Op*. Os campos Rt, Rs1, Rs2 indicam respectivamente registrador destino e registradores fonte.

O mesmo ocorre com os *flags*, pois as instruções podem afetar *flags* diferentes. Portanto, cabe ao usuário informar quais são os *flags* afetados em cada instrução.

O código objeto é limitador do número de instruções. A cada instrução inserida no configurador é analisado seu código objeto para que nenhuma outra instrução possa utilizar o mesmo código, limitando assim o número de instruções que podem ser representadas.

Ao encerrar a configuração da arquitetura é criado o arquivo "*nome_processador*.txt", contendo a descrição da arquitetura que é utilizada para modelar o montador e o simulador para que os mesmos aceitem as aplicações em *assembly* escritos para a arquitetura parametrizada.

5 Ambiente de montagem e simulação

Uma importante contribuição deste trabalho é o ambiente de montagem/simulação parametrizável, permitindo a geração/validação de diferentes arquiteturas.

O simulador, desenvolvido em linguagem Java, possibilita a simulação de programas descritos em *assembly*, para a arquitetura definida pelo configurador descrito na Seção anterior. O arquivo de configuração determina o nome da arquitetura, quantos registradores, os *flags*, o tamanho da memória, o funcionamento de pilha e as instruções que fazem parte da arquitetura. Itens como os registradores IR, PC e SP são fixos, com comportamentos independentes da arquitetura descrita. Outros itens fixos no simulador são as tabelas da memória e símbolos, que podem variar apenas em seu tamanho.

A Figura 6 ilustra a interface gráfica do simulador. A esquerda desta figura está apresentada a tabela de memória, contendo em cada linha a instrução em *assembly*, o endereço da posição da memória e o código objeto. Ao centro é

inserida a tabela de símbolos, onde são apresentados o nome do símbolo, seu endereço de memória e o seu valor. A direita da figura estão localizados os registradores de uso geral e os registradores IR, PC e SP. Na parte inferior são ilustrados os botões de controle *Step*, *Run*, *Pause*, *Stop* e *Reset* e as opções de velocidade *Lento*, *Normal* e *Rápido*. Os qualificadores de estado encontram-se na parte inferior à direita.

Figura 6 – Janela do simulador.

A ferramenta de montagem é transparente para o usuário, pois a mesma está integrada ao simulador como um método nativo [18]. A interface com usuário é apenas a interface do simulador. Um trabalho a ser desenvolvido para este ambiente integrado é um editor de texto para a descrição *assembly*. Esta integração ocorre devido ao fato de que o simulador não processa instruções diretamente em *assembly*, mas sim em código objeto. O montador, escrito em linguagem C, é responsável por converter uma aplicação em *assembly* para o código objeto, determinado na configuração da arquitetura. Para realizar a conversão de uma aplicação em *assembly* para código objeto, o montador passa por três fases: análise sintática, substituição de texto e criação de arquivos.

- A fase de análise sintática consiste em verificar se as linhas de instruções correspondem às instruções descritas pela arquitetura, ou seja, identifica qual instrução está sendo referenciada, quantos registradores, quantas variáveis e quantos *labels* são relacionados à mesma. Nesta fase de análise sintática é criada a tabela de símbolos, as variáveis e as constantes com seus endereços respectivos. Quando é encontrado um identificador não presente na tabela, este é chamado de *pseudo-label*, definição dada às palavras que podem estar declaradas mais adiante no programa.
- A fase de substituição de texto consiste em substituir *labels*, *pseudo-labels, variáveis e constantes*, por seus respectivos valores binários, respeitando o tamanho da palavra reservado a eles. Por exemplo, suponha a instrução **LDH R1,#x** (carregar a parte alta do registrador **R1** com o conteúdo da parte alta da constante "**x**") e que seu código objeto conforme o arquivo de configuração seja "**7 Rt Cte Cte**" (opcode 7, seguido de registrador destino e dois campos de constantes de 8 bits). Se a constante "x" for 10A3, o código objeto final será "**7110**". O número de registradores pode variar de 1 a 16 conforme a arquitetura descrita, e são representados em *assembly* por R*x*, onde *x* é um natural entre 0 à 15. No código objeto os registradores são substituídos por seus valores em hexadecimal. Por exemplo, supor a instrução **ADD r1, r9, r13** e o código objeto "**0 Rt Rs1 Rs2**", logo o código objeto final será "**019D**".
- A terceira etapa do montador é responsável pela criação de três arquivos: (*i*) "hex" utilizado para fazer *download* do código objeto na memória da plataforma de prototipação; (*ii*) "sym" utilizado pelo simulador; e (*iii*) o arquivo opcional "txt" utilizado no *test_bench* de simulações VHDL do processador e pelo usuário para verificar o código montado para cada instrução.

Erros encontrados durante a execução de uma das fases do montador são salvos em um arquivo de mensagens. Este arquivo é lido pelo simulador após a execução do montador afim de que os erros sejam apresentados ao usuário e não se prossiga a simulação. Erros na execução do montador não possibilitam a simulação, porque os mesmos indicam que as instruções da aplicação *assembly* não condizem com as instruções existentes na arquitetura.

A capacidade de simular instruções de diferentes arquiteturas é permitida através de um conjunto de instruções pré-definidas que são associadas às instruções definidas pelo configurador de arquiteturas, conforme explicado anteriormente. Esta associação de instruções, mesmo que possuam nomes diferentes, é que permite a simulação. Por exemplo, é possível associar o *mnemônico* ADIÇÃO à instrução pré-definida ADD, e desta forma permitir ao simulador encontrar o código objeto da instrução ADIÇÃO, para que o mesmo seja interpretado e executado como sendo a instrução ADD.

Como as instruções podem afetar qualificadores diferentes e independentes de qualquer outra instrução, cada instrução contém uma lista dos qualificadores que são afetados com a sua execução. Esta lista também está presente no arquivo de configuração e é feito acesso a esta ao final da execução de cada instrução para que os qualificadores sejam atualizados conforme a mesma.

As instruções que fazem acesso a pilha dependem do registrador SP, que é inicializado com valores diferentes dependendo do funcionamento da pilha. Se a pilha funciona com pós-incremento, a posição inicial da pilha será o endereço de memória que corresponde a 90% do tamanho da memória (escolha *default*, podendo ser alterada na configuração do processador). Caso a pilha seja DEC, a posição inicial da pilha será o último endereço de memória. A posição inicial da pilha também pode ser alterada pela instrução LDSP, independentemente de seu funcionamento.

O controle da simulação é realizado através dos botões *Step*, *Run, Pause*, *Stop* e *Reset*.

6 Conclusão e Trabalhos Futuros

A contribuição deste trabalho é disponibilizar ao projetista um processador flexível, que pode ter seu conjunto de instruções adaptado à aplicação a que o processador executa. Este processador serve para prover uma forma simples e eficiente de executar software em SOCs. Além disto servirá no contexto de futuros trabalhos em hardware/software codesign.

Dois trabalhos relacionados encontram-se hoje em desenvolvimento. O primeiro é a implementação do compilador C para a arquitetura parametrizável R6 e o segundo é o gerador automático de processador dedicado (ASIP), em função da aplicação do usuário e de uma base de dados com os módulos funcionais básicos do processador, descritos em VHDL.

Agradecimentos

Os autores Fernando Moraes e Ney Calazans agradecem o suporte do CNPq (projetos integrados número 522939/96-1 e 522939/96-1) e da FAPERGS (projetos número 96/50369-5 e 94/01340-3).

Referências

[1] DE MICHELI, G. - **Hardware/Software Codesign: Application Domains and Design Technologies**. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, Italy, 1995.

[2] LAVAGNO, L., SANGIOVANNI-VICENTELLI A. and HSIEH H. - **Embedded System Codesign: Synthesis and Verification**. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, City Italy, 1995.

[3] SLOMKA, F., DORFEL, M., MUNZENBERGER, R. and HOFMANN, R.- **Hardware/Software Codesign and Rapid Prototyping of Embedded Systems**. IEEE Design & Test of Computers, April 2000.

[4] SCIUTO, Donatella - **Guest Editor's Introduction: Design Tools for Embedded Systems**. IEEE Design & Test of Computers, April 2000.

[5] GUPTA, R., COELHO Jr. and DE MICHELI G. - **Program Implementation Schemes for hardware-software systems**. IEEE Computer, pages 48-55, January 1994.

[6] ERNST, R. and HENKEL, J. - **Hardware/Software Codesign of Embedded Controllers Based on Hardware Extraction**. In Proceedings of the International Workshop on Hardware/Software Codesign, 1992.

[7] BERRY, G. - **The ESTEREL v5_21 System Manual**, available at homepage: http://www.esterel.org. INRIA, 1999.

[8] BARROS, E. S. - **Hardware/Software partitioning using UNITY**. Ph.D. thesis Tübingen University, Germany, 1993.

[9] PATT, Y. N., PATEL, S. J., EVERS, M., FRIENDLY, D. H., STARK, J. – **One Billion Transistors, One Uniprocessors, One Chip**. IEEE Computer, pages 51-57, September 1997.

[10] HAMMOND, L., NAYFEH, B. A., OLUKOTUN, K. - **A Single-Chip Multiprocessor**. IEEE Computer, pages 79-85, September 1997.

[11] SCHALLER, R. – **Moore's law: past present, and future**. IEEE Spectrum, 34(6):52-59, June 1997.

[12] ALTERA, Inc. – **Excalibur Series FPGAs**. available at homepage: http://www.altera.com/html/products/excalibursplash.html. USA, 2000.

[13] Hennessy, John, Patterson,David - **Computer Organization and Design: the Hardware/ Softwarwe Interface**, Englewood Cliffs, 1994.

[14] Ney Calazans, Fernando Moraes. **VLSI Hardware Design by Computer Science Students: How early can they start? How far can they go?** In: *1999 Frontiers in Education Conference*, San Jose, PR, pp. 13c6-12 to 13c6-17, November, 1999. Available at: http://fairway.ecn.purdue.edu/~fie/.

[15] MAZOR, S., LANGSTRAAT, P - **A Guide to VHDL**. Kluwer Academic Publishers, Norwell, MA, 1992.

[16] F. MORAES, N. CALAZANS, E. FERREIRA, D. LIEDKE. **Implementação eficiente de uma arquitetura load/store em VHDL**. CORE 2000, p.2-13.

[17] Active HDL Simulator. Available at http://www.aldec.com

[18] MORGAN, MIKE -**Using Java 1.2**. QUE, pages 82-86, June 1998.